Approved for public release; distribution is unlimited.

LARGE GRAIN DATA-FLOW GRAPH RESTRUCTURING FOR EMSP SIGNAL PROCESSING BENCHMARKS ON THE ECOS WORKSTATION SYSTEM

by

David P. Swank
Lieutenant, United States Navy
B.A., East Stroudsburg State College, 1982

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
June 1993

Michael A. Morgan, Chairman,
Department of Electrical and Computer Engineering

UNCLASSIFIED

ITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

EPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
ECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution is unlimited.
ECLASSIFICATION/DOWNGRADING SCHEDULE	
RFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
AME OF PERFORMING ORGANIZATION: ctrical Engineering Dept. val Postgraduate School; 6b. OFFICE SYMBOL (if applicable): EC	7a. NAME OF MONITORING ORGANIZATION: Naval Postgraduate School
DDRESS (City, State, and ZIP Code): nterey, CA 93943-5000	7b. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943-5000
AME OF FUNDING/SPONSORING RGANIZATION; 8b. OFFICE SYMBOL (if applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
DDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

TITLE (Include Security Classification): LARGE GRAIN DATA-FLOW GRAPH RESTRUCTURING FOR EMSP SIGNAL PROCESSING ENCHMARKS ON THE ECOS WORKSTATION SYSTEM (U)

ERSONAL AUTHOR(S): Swank, David P.

TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
ster's Thesis	FROM TO	June 1993	163

UPPLEMENTARY NOTATION: The views expressed in this thesis are those of the author and do not reflect the official olicy or position of the Department of Defense or the United States Government.

COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
ELD	GROUP	SUB-GROUP	Revolving Cylinder, Parallel Processing, Data-flow, EMSP, Graph restructuring, Trigger Queues,

BSTRACT (Continue on reverse if necessary and identify by block number)

his thesis documents a procedure for implementing the Revolving Cylinder scheduling algorithm for parallel pro- ms on the ECOS Workstation System (EWS), designed specifically by AT&T for simulation of the Enhanced dular Signal Processor (EMSP) currently in use by the United States Navy. The Revolving Cylinder (RC) algo- m provides a methodology for forcing First Come First Served (FCFS) schedulers to follow a more systematic ization of available resources. The methods of implementation used take advantage of the Graphical Editor (*gred*) nsert additional data dependencies into the program structure. The thesis utilizes applications written in Signal Pro- sing Graph Notation (SPGN), viz., a simple correlator function and the active subroutine of the U.S. Navy noobuoy benchmark. Results for standard FCFS scheduling and RC modified scheduling are presented for both ecial attention is paid throughout the thesis to enhancement of manufacturer supplied documentation with regard mplementation of the non-standard RC structures. Impact of the algorithm on throughput and latency is discussed vell as performance determination using the tools provided with the ECOS Workstation System.

ISTRIBUTION/AVAILABILITY OF ABSTRACT: ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION: UNCLASSIFIED
NAME OF RESPONSIBLE INDIVIDUAL: idhar B. Shukla	22b. TELEPHONE (Include Area Code): (408) 656-2764 / 22c. OFFICE SYMBOL: EC/Sh

ORM 1473, 84 MAR — 83 APR edition may be used until exhausted; All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE: UNCLASSIFIED

Abstract

This thesis documents a procedure for implementing the Revolving Cylinder scheduling algorithm for parallel programs on the ECOS Workstation System (EWS), designed specifically by AT&T for simulation of the Enhanced Modular Signal Processor (EMSP) currently in use by the United States Navy. The Revolving Cylinder (RC) algorithm provides a methodology for forcing First Come First Served (FCFS) schedulers to follow a more systematic utilization of available resources. The methods of implementation used take advantage of the Graphical Editor (*gred*) to insert additional data dependencies into the program structure. The thesis utilizes applications written in Signal Processing Graph Notation (SPGN), viz., a simple correlator function and the active subroutine of the U.S. Navy Sonobuoy benchmark. Results for standard FCFS scheduling and RC modified scheduling are presented for both. Special attention is paid throughout the thesis to enhancement of manufacturer supplied documentation with regard to implementation of the non-standard RC structures. Impact of the algorithm on throughput and latency is discussed, as well as performance determination using the tools provided with the ECOS Workstation System.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

I. INTRODUCTION

Early detection and identification of possible threats have been of paramount importance to naval fleets throughout history. This has evolved from the lookout in the "crow's nest", to the sophisticated electronic sensors of today's Navy. Platforms at sea must be able to process data in ever-increasing amounts and complexity. In addition, the speed with which this data must be processed is also increasing geometrically.

Signal processing requirements for the Navy in 1990 ranged from 300 million floating point operations per second (MFLOP) for small airborne sensors, to 2.4 million MFLOPs for submarine sonar arrays [Ref. 1:p. 2]. The proliferation of highly automated systems and 'smart' weapons has led to the requirement for a programmable, reliable signal processor with high throughput and inherent scalability. Although processing power of individual components has increased significantly over recent years, the physical limits of technology require a departure from traditional von Neumann architectures.

To meet these needs through the 21st century, the United States Navy has developed the AN/UYS-2 Enhanced Modular Signal Processor (EMSP).

A. BACKGROUND

Until the advent of the AN/UYS-2, all signal processors in the U.S. Navy utilized time-line control-flow architectures. Here a series of instructions and corresponding data are processed sequentially and initiated by a single control signal. While multi-threaded control-flow systems are efficient for certain processing tasks, multiple data stream applications are difficult to write [Ref. 2:p. 25].

Execution in a data-flow architecture, however, can occur asynchronously when hardware resources and data become available, and is independent of control signals. This contrasts with control-flow in two important ways. Data exists only between production and consumption by an executable entity, and instructions are not fetched sequentially from a memory stack using a program counter. Data-flow naturally supports concurrent processing, but suffers from high communication and bookkeeping costs [Ref. 3, Ref. 4].

Data-flow implementations can be classified from fully dynamic to fully static [Ref. 5:p. 334] based on the amount of *a priori* information available to the compiler. Fully static scheduling requires a deterministic and data-independent program, but requires minimal overhead since all execution events are set at compile time.

At the other end of the spectrum is fully dynamic data-flow scheduling. Here scheduling of an executable entity simply demands that the input operands are present and that a processor is available. The scheduler is acting only as a data dispatcher, matching data, instructions and processors on an *as available* basis [Ref. 6]. This approach can maximize use of resources and fully exploit the concurrency of data-flow architectures, but overhead requirements can be prohibitively expensive [Ref. 5:p. 335].

One solution to maximize the concurrency inherent in data-flow architectures, while minimizing the associated overhead, is to use a hybrid architecture. Such an architecture would take advantage of control-flow for the task level, and data-flow at the functional level. This architecture is incorporated into the AN/UYS-2.

1. EMSP Design

The modular, programmable design of the EMSP utilizes a multiprocessor based, hybrid data flow architecture capable of high throughput with acceptable overhead costs. Modularity is achieved through the use of Standard Electronic Module (SEM) based technology to form the six Functional Elements (FE) comprising the EMSP. This provides a reliable, configurable system with high output and reduced software and maintenance costs.

The distributed run time operating system implemented in the AN/UYS-2 uses control-flow at the node (task) level and large grain data flow (LGDF) at the graph (functional) level. This increase in data granularity significantly decreases communication and bookkeeping costs, while exploiting the parallelism typical of signal processing applications.

Signal processing applications map naturally to data-flow graphs. In turn, these graphs consist of many pre-defined functions which may be programmed once at the assembly language level, then incorporated into the graph structure as needed. The pre-defined functions, or primitives reside in a library until loaded into the Global Memory (GM) modules at compile time.

The operating system is complemented by the modularity of the SEM-based hardware implementation, allowing additional resources to be incorporated without application code modification.

2. Design Problems

EMSP uses a First-Come-First-Served (FCFS), non-deterministic scheduling algorithm. Under high loads, this may create excessive processor pending times and overhead costs resulting in unpredictable arrival of processed data [Ref. 6]. A side issue of the unbalanced flow of data through the graph is large peak memory requirements due to queuing.

Any compile-time mechanism which addresses this issue should modify existing code using well documented techniques. It must also be validated by thorough test simulations over the expected range of operating extremes.

B. OBJECTIVES

The Revolving Cylinder algorithm allows controlled data-flow execution, which increases predictability and improves some aspects of performance. This thesis investigates the procedures required to implement the Revolving Cylinder (RC) scheduling algorithm using the ECOS methodology including:

- Elaboration of techniques in the existing ECOS literature with emphasis on those areas most relevant to restructuring
- Modification to a simple, non-hierarchical graph structure using RC
- Modification of the active and passive portions of the Navy developed sonobuoy benchmark graphs using the Revolving Cylinder algorithm

• Discussion of the simulation and performance analysis tools available under ECOS

•Performance analysis of the restructured graphs

The techniques utilized are simple modifications during application development and have no impact on the run-time mechanisms.

C. THESIS ORGANIZATION

Chapter II discusses the ECOS methodology in detail. The interrelation between ECOS and EMSP is explained and components are explained in detail. Actual implementation of a simple graph is detailed for illustrative purposes. Chapter III discusses use of the EWS suite to characterize signal processing graphs and provide the basis for enhancement analysis. Chapter IV introduces the Revolving Cylinder scheduling approach and discusses the theory of implementation under ECOS. Modifications to the graph structure and the associated code are discussed. The example graph implemented in Chapter II and the Navy sonobuoy benchmark are used as sample structures. Simulation procedures and performance analysis are explained in Chapter V. Comparisons of the baseline and restructured graphs are made. Chapter VI provides a summary of the work, along with recommendations for further investigations.

II. ECOS FUNDAMENTALS

The distributed run-time operating system and hardware design of the AN/UYS-2 support a specialized data-flow software using the Navy developed Processing Graph Methodology (PGM). Using EMSP Common Operational Software (ECOS), signal processing applications are generated at workstations using a graphical interface to construct 'graphs' representing the desired application and its associated data flow. This graphical representation is then translated into Signal Processing Graph Notation (SPGN) which can be utilized by the EMSP.

The system utilized for all work in this thesis is the ECOS Workstation System (EWS) Version 5.5, written by AT&T Bell Laboratories [Ref. 7].

A. DATA-FLOW REPRESENTATION

Signal processing applications, by their nature, map easily to Large Grain Data-Flow (LGDF) graphs. The typical progression between transforms, filters and other Digital Signal Processing (DSP) functions, is represented as nodes in the graph. These nodes correspond to atomic segments of code which must execute sequentially. Since no attempt is made to exploit concurrency within a node, its complexity governs the granularity of the program [Ref. 8:p. 22].

Between each functional piece of code is a storage area termed a queue or edge. Data is consumed from a queue when a match is made between an available processor and the node which the queue feeds. This node represents the head of the queue, while the source node represents the tail. Information is deposited on, and removed from, queues on a First In, First Out (FIFO) basis. The ability of these queues to store data from multiple firings of the source node decouples the dependencies between nodes on either end of the queue. This is crucial to permit concurrency [Ref. 1].

Another important feature of this approach is the ability to consolidate graphs which perform a desired function into a subgraph. These may be used in a hierarchical manner to perform the overall application.

The nodes and subgraphs defined within the ECOS framework require that *threshold*, *read* and *consume* amounts for the incoming data and the *produce* amounts for the output be specified at compile-time.

Since these designations have slightly different connotations in other areas of computer science, they will be defined here. The threshold amount is the quantity of data which must be present on the providing queue in order for the node to execute. Once this is reached, the node is considered ready for execution by the scheduler and is assigned a processor based on the scheduling algorithm. Read amount is that quantity which is read in for use in the current execution of the node. This defaults to the threshold amount if not specified, but can be modified if required. The consume amount is the amount the queue is decremented following a read operation. This again defaults to threshold quantity, but is user definable. The produce amount is the amount of data placed on the receiving queue after execution of a node.

It is important to note here that EWS does not actually process data. Instead, it simulates data flow based on known parameters, graph topology and hardware resources. The known parameters include input and output channels available, data rates, queue threshold and initialization amounts, node read and consume amounts, node production quantities, and various other user-defined quantities.

B. HARDWARE CONSIDERATIONS

Six functional elements (FE) comprise the working components of the AN/UYS-2. Each is separately constructed and self-supporting in hardware and software requirements. This enhances the modularity of the system, and prevents a single failure from halting the system. The FEs are the Arithmetic Processors (AP), Global Memory modules (GM), the Command Program Processor (CPP), the Input/Output Processors (IOP), the Scheduler (SCH), and the Input Signal Conditioner (ISC). Additional components are possible and may be incorporated at a later date [Ref. 2].

Figure 2.1 is a diagrammatic representation of the virtual EMSP seen by the EWS system. The CPP and ISC are not utilized for simulation purposes.

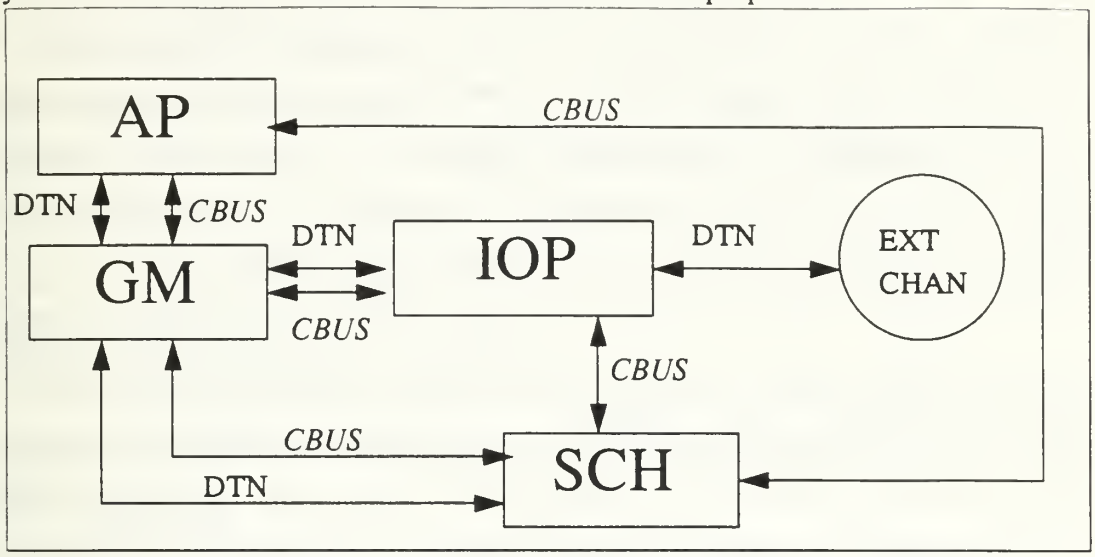


Figure 2.35: Virtual Machine Schematic

Two of the FE functions are transparent to EWS operation. The CPP functions are performed by the EWS host machine processor during preparations for simulation. The ISC functions are not applicable to EWS, since no actual data is processed. The reader is referred to [Ref. 2] for further discussion of these functions.

Two separate networks support communication between FEs. The Data Transfer Network (DTN) supports data flow between the IOP, AP and GMs. Control information, data requests and test functions are supported by the Control Bus (CBUS). Configurations with 2, 4, 8 or 16 ports are possible for the DTN. Each port on the backplane is also expandable to permit up to four FEs per port using distributors and concentrators. These constraints are maintained in the EWS virtual machine. The 32 bit DTN bus is capable of supporting up to 16 simultaneous transfers [Ref. 2].

The Input/Output Processor (IOP) transfers raw data to and processed data from the Global Memories via the DTN. Each is capable of up to 5 MHz maximum bandwidth, supporting a 16-bit word format on 16 channels. Only the 16 channel limitation must be observed in the EWS implementation normally.

Global Memory (GM) modules are configurable with up to 32 megabytes (Mbytes) of Dynamic Random Access Memory (DRAM). GMs store data and node execution instructions, and are responsible for monitoring queue threshold levels. Once sufficient data has accumulated on a queue, GM notifies the Scheduler that the queue has exceeded threshold. Consumption of a queue allows the GM to free the allotted memory space for future use. Node Execution Parameters (NEP) and instructions are passed to the Arithmetic Processors by the GM in response to signals from the Scheduler [Ref. 2].

The Scheduler (SCH) is implemented on SEMs identical to Global Memory. DRAM memory is usually limited to 2 Mbytes. As information is received from the GMs that queues are over threshold, and from APs that nodes are finished processing, the Scheduler matches available resources (AP) to ready nodes on a First Come, First Served (FCFS) basis. Four tables are maintained by the Scheduler as a database for this matching procedure.

Actual signal processing is performed on the Arithmetic Processors (AP). The AP is divided into the Control Unit (CU), Address Generator (AG), and Arithmetic Unit (AU), each implemented on a separate SEM module. The Arithmetic unit has four parallel pipelines, each provided with a floating point multiplier and two adders [Ref. 2]. The multiple pipeline arrangement provides the application programmer with increased development flexibility.

The AN/UYS-2 uses two Standard Electronics Module (SEM) configurations, SEM B and SEM E. The former operates at a 7 MHz clock rate and uses 44 modules for the basic implementation [Ref. 2]. The later uses a 10 MHz clock, uses only 10 modules for the basic implementation and has several other cost and weight advantages. EWS supports use of both configurations via a flag in the simulation command line.

C. EWS TOOLS

The ECOS Workstation system provides a number of tools for graph creation, translation into Signal Processing Graph Notation (SPGN), compilation into object code

for the EMSP or simulation, library reference tools, and others which provide the application developer a convenient, powerful environment.

Graph topology is implemented using the Graphical Editor (*gred*), which permits an object-oriented style of code creation. Once the graph has been created using *gred*, it is converted to SPGN using the *grail* translator. This converts the graphical representation into compilable code, while checking for continuity and syntactical errors. Also provided are three programs which validate and compile the SPGN; *grasp*, *glitr* and *cpcc*. These are combined in a supplied script file named *ggcc*.

Grasp translates the SPGN source code into graph tables. *Glitr* translates the tables in C language routines and *cpcc* compiles the C routines and command program, then links all the programs [Ref. 7:p. 7.2-7.4].

Once an executable file is created, static and dynamic simulations may be accomplished using the Graph Analysis, Static (*gas*) and Event Time Simulator (*ets++*) functions. An additional graphical interface, Simulator Performance Interface (*spi*) is provided to monitor the dynamic simulation. Figure 2.2 is a flow diagram of application development using ECOS and the interfaces to EMSP.

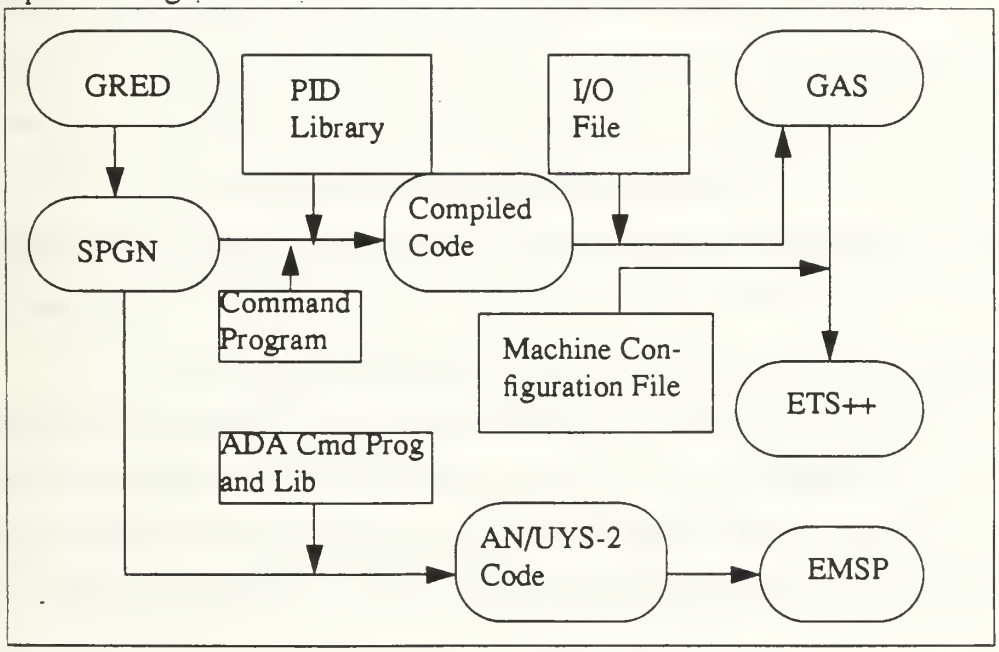


Figure 2.2: ECOS Methodology Design Flow

Several ancillary programs help in graph creation (*primr*), machine configuration file editing (*med*) and execution analysis (*tbr*). Each of these will be elaborated on later.

1. Graph Creation and Editing

Gred is operable only under an X-windows environment such as Openwindows Version 3 currently installed on the Electrical and Computer Engineering Department Sun workstations. Basic procedures for graph creation are delineated in [Ref. 7:p. 3.1-3.6] and [Ref. 9:p. 1-32], however, they are very brief and elementary. Anyone attempting to use *gred* for the first time is strongly urged to utilize the ECOS Tutorial [Ref. 9]. Familiarity with the Processing Graph Method (PGM) Tutorial [Ref. 10] is assumed.

Step one in graph creation is to place the graph elements in the desired topological pattern. The topology of the graph is primarily determined by the application and node primitives chosen to implement the program. Five graph elements are mandatory for any application: an input queue, an executable node, an output queue, and the two interconnections termed ports.A relatively simple topology, but one which incorporates most of the structures commonly utilized, is shown in Figure 2.3 below.

The topmost hexangonal structures are the input queues, which provide an interface between the command program and the application program itself. Each of these queues is allotted the larger of 32 kilobytes, or 8 times the threshold value assigned, in global memory space. The command program is a separate program from SPGN, which serves only to pass input and output handling instructions and some formal parameters, if used, to the SPGN program. Formal parameters provide a method for modifying SPGN constants and variables without recompilation of the SPGN code.

Nodes are represented by the circular structures. Each node primitive (PRIM) must have an associated PID and entry in the *pidtoc* (PID table of contents). Actually, an entry in the pidtoc is all that is required to create an executable program. However, without the associated PID, no execution times or produce amounts are assigned to the node. The

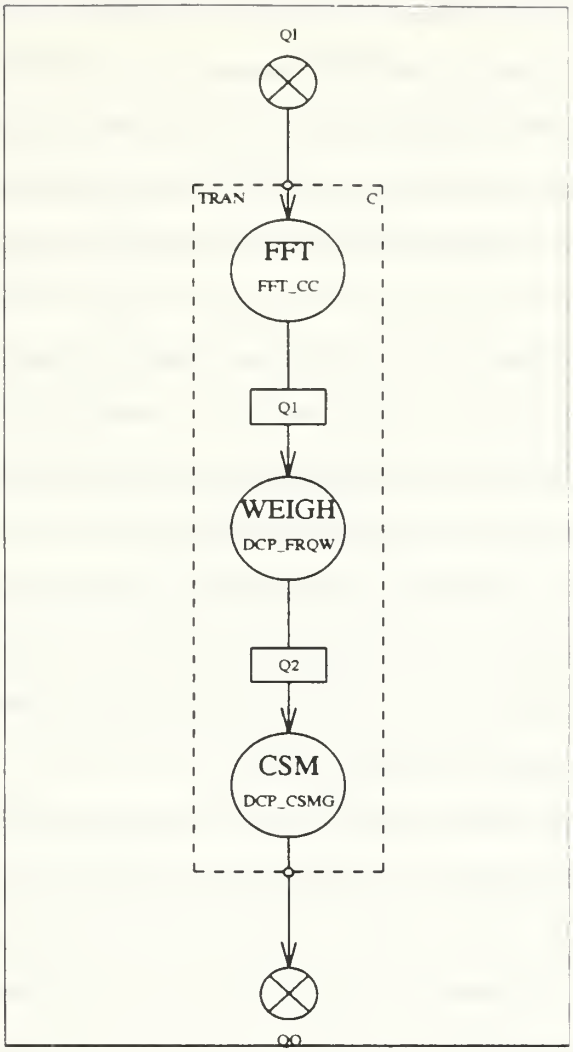


Figure 2.3: Gred produced sample graph

pidtoc and its associated PIDs must reside in the *PIDS* subdirectory under *Ecosws*, the main EWS directory.

The lower hexagonal structures are the output queues and are the counterparts of the input queues discussed above. A point to note during graph creation is that for each word of data written to the input queue by the IOP, a corresponding word must be consumed by the IOP from the output queue. This sounds intuitively obvious, but some nodes alter the data sizes during execution, and so threshold, read, consume and produce amounts must match data flow or the graph will execute erroneously.

Some additional structures, not mandatory, but always found in any usable graph are shown on Figure 2.3. Queues located between nodes are referred to as local queues, and are allotted four kilobytes of global memory, or four times the threshold value of the receiving node, whichever is larger [Ref. 1:p. 4.1]. Understanding of the correct use of local queues and their role in data flow is paramount in successful use of the EWS system.

The two *kinds* of local queues are the single and family structures. Generation of family structures is possible for almost all graph elements. The primary advantages of family structures are aesthetic, reduced clutter on the workspace and less code from the translation function *grail*. The primary disadvantage is flexibility. Once a family structure is created, all members of the family are treated identically. This means that any data type intended for one member must also be presented to the remaining family members. This fact has significant consequences for implementation of the Revolving Cylinder algorithm, as discussed in Chapter IV. Family members are differentiated by the *mselector* parameter, definable during the editing of the element [Ref. 7].

Local queues act as the buffer between two nodes, each of which may have a family structure. This may represent convergence or divergence in the data flow of the graph, and care must be taken in proper selection of local queue and port structure to support the required flow. The table at the end of Appendix A of the User's Manual [Ref. 1:A-26] must be followed exactly.

Ports, or connections, as mentioned above, provide the capability for creating a divergence or convergence of data with the graph structure. Single ports carry the data from the source node to the receiving queues as determined by the graph topology. If the source node is of type family, then a single port will carry information from a single member to either a single queue structure, or to a single member of a family queue structure. This one to one correspondence between ports and their associated queues and nodes plays a major role in implementing Revolving Cylinder algorithm- based restructuring.

The pre-defined values which each node expects prior to execution may be passed in one of several ways. Data is always passed via the port connections, through queues, to

other nodes or to the I/O processor via output queues. Additional information can be obtained from Graph Initialization Parameters (GIP) or by Graph Variables (GV). These are shown as boxed quantities, usually to the upper left of the actual graph structure. Graph variables have a similar appearance on the GRED interface. Information to be passed using GIP or GV may be represented by literals, arithmetic functions, variables, or a combination of these.

All data must be identified during editing of the node, and is entered as Node Execution Parameters (NEP). They must be entered in the order expected by the PID, and may take the form of literals, the name of the GIP, GV or associated queue, or as macros. The two most often used macros are $INx and $OUTx, where x is an non-negative integer. Again, macros are interpreted from left to right as depicted in the node representation appearing in the editing window. Failure to match NEP macro ordering to the position assigned by the program will create errors and the routines will not compile properly.

Since referral to the PID library during each editing session would be extremely tedious, the EWS provides a pop-up graphical representation of the PID for comparison and selection purposes using the *primr* function. This is not available from within *gred* and must be invoked from a separate window.

Increasing read amount increases the number of AP cycles per node and decrease the number of nodes scheduled per second. This may lead to better system efficiency, but increases latency and may require more GM memory. Decreasing the produce amount, increases the scheduling rate, decreases the average transfer amount and decreases graph latency. Although these are useful methods to accomplish desired performance enhancement, care must be taken not to violate constraints imposed by the primitives involved [Ref. 4:p. 1].

Custom PIDs may be written to modify a preexisting PID for a particular application, to perform a function not incorporated in the current PID library, or to create a single executable entity from a group of nodes (chain) [Ref. 7:p. 5.2]. When writing a PID for a chain, the inputs must match the inputs of the first node in the chain (chains must begin

and end on nodes and cannot contain input or output queues), and the outputs must match the outputs of the last node in the chain. The new executable entity, is now viewed as a single node by the scheduler. This method of enforcing execution of several sequential nodes on a particular processor can be exploited in the Revolving Cylinder paradigm, as well as other optimization techniques [Ref. 12].

Introduction of dependency arcs to a data-flow graph should not require modifications to existing PIDs. The chaining approach to scheduling enforcement is beyond the scope of this thesis, and the reader is referred to [Ref. 11] for further information on PIDs and the PID library.

2. Hierarchical Graph Construction in ECOS

The ability to build higher order graphs from smaller graphs provides a powerful tool for generating applications. This also permits simplified debugging since each subgraph may be individually translated and compiled. Other than the subgraph icon shown in Figure 2.4 below, no additional structures are required to construct a hierarchical structure in ECOS.

The notations in the upper left of the figure are the Graph Instantiation Parameters (GIP) and Global Variables (GV) for the graph. The vertical bar to the left denotes a GV, and the horizontal bars alone indicate a GIP.

Data is passed between levels in the hierarchy using input and output queues. When a subgraph is connected via a port to a queue on one level, an input or output port of the same *kind* as the port (i.e., single or family) is inserted at the next lower level in the hierarchy. The subgraph, which is a complete graph structure itself, is then connected to these *actual* input and output queues. The resulting appearance is that of a double input or output queue arrangement at the subgraph level.

Families of subgraphs are also possible. As with all other family structures, some mechanism must exist to indicate the number of family members. This mandates the use of a local index for the subgraph family. An example of a local index is "I = 1..NB", where

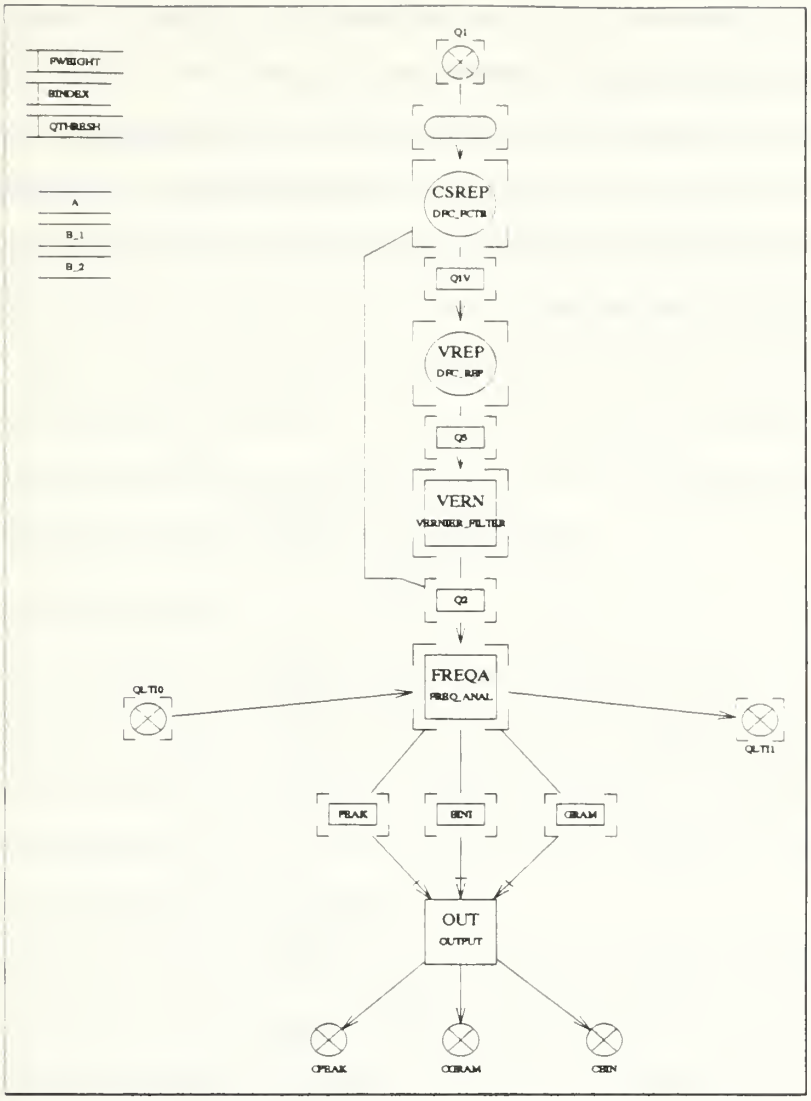


Figure 2.4: Passive Sonobuoy Root Level graph

"I" is the local index and "1..NB" indicates family members numbered from 1 to NB, which is a predefined constant (GIP). Whenever a local index is used for a family construct, queues and ports connected to it **must** use the local index is defining their mselectors.

Graph Instantiation Parameters (GIP) and Graph Variables (GV) are not global in scope. Parameters required by the subgraph are passed to it by means of the subgraph icon. Editing this will pass parameters of the type specified (GIP or GV) in the order listed. At the subgraph level, these appear as formal GIPs and formal GVs. They must be declared as

formal parameters in the order passed, with name and mode specified. Arbitrary names may be assigned here independent of those used at the higher level. If local parameters are needed, they may be declared for each subgraph and will not be seen by other subgraphs. The vernier filter subgraph, seen as the topmost subgraph on Figure 2.4, appears below as Figure 2.5. Two local GIPs are indicated, along with the input and output queues. Actual input and output queues are not shown.

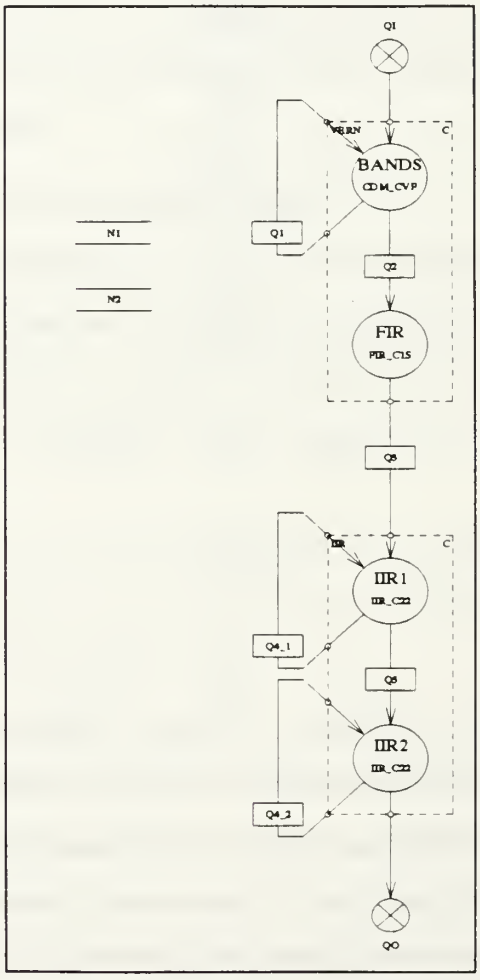


Figure 2.5: Vernier Filter subgraph

3. SPGN Generation

Once the graph is created, the translation into SPGN is performed by using the *grail* function. *grail* must be invoked from a command line (i.e., not from within *gred*), and

the -g suffix flag is normally used to sort the graph GIPs and GVs to ensure they are declared in the proper order. If the graph elements are topologically correct (proper use of port and queue types, data modes consistent from element to element, etc.), *grail* will normally execute without difficulty.

A typical error is mode mismatches, where data is declared as being of type integer, float, and so forth, but is passed to a structure, such as a queue, which is expecting another data type. At this step in the process, the mode mismatches are between elements in the command program and SPGN. *Grail* errors can often be corrected by editing the SPGN, but it is recommended that corrections be made to the original graph to prevent reoccurrences if the graph structure must be modified later.

4. SPGN Compilation

After the graphical representation has been translated to SPGN, compilation into an executable file may begin. At this point, the command program must be provided to supply the I/O information and formal parameters required. Compilation is usually accomplished in two steps, use of the *ggcc* script file to produce an object file, and conversion to an executable simulation file.

The former actually encompasses three separate functions mentioned previously, *grasp*, *glitr*, and *cpcc*. Each of these may be invoked independently, rather than using the *ggcc* script file. For more on this, please see the ECOS User's Manual [Ref. 7:p. 7.1-7.3].

Conversion of the object code produced from the first step to an executable file is done with a simple command line statement 'objectfile > *.ets', where the '*' can be any legal filename, and 'objectfile' is the output of *cpcc*. The '.ets' suffix identifies the file as an executable simulation file. It is at this point that the PID library is linked, execution times assigned and assigned produce, threshold, read and consume amounts are checked to ensure proper data flow. Typical errors again stem from mode mismatches created during re-editing, or from improper assignment of ports and queues.

Once this point is reached, the program is normally error free and ready for simulation. Situations can arise where errors may not be detected until simulation is attempted. Since only element modes and types are checked during the above processes, errors in actual numbers of structures may not be detected until simulation is attempted. For instance, if a node family of size five is feeding a queue family of size ten through a port family of size ten, all element types and modes may be correct, but five queues will have unlinked 'tails' due to the one to one correspondence convention.

5. Simulation using *gas*

Once a completely error-free program is obtained, static and dynamic simulation may commence. The static graph analyzer (*gas*) and event time simulator (*ets++*) also require a separate input/output (*.io) file which contains data rates, data block sizes and threshold, consume, read and produce amounts for the output and input queues as needed. Examples of input/output files for the baseline correlator and benchmark graphs may be found in Appendices G (correlator graph) and I (benchmark graph).

Gas provides a best case analysis of the program performance based on unlimited hardware resources and supplied data rates. Contention for resources and waiting times are not considered here, since the purpose of static analysis is to provide enough information to derive a hardware configuration which will provide sufficient resources to minimize these problems. The User's Manual [Ref. 7:p. 9.1-9.13] and ECOS Tutorial [Ref. 9:p. 40-44] contain comprehensive explanations of each of the results obtained using *gas*.

With the information provided by *gas*, the hardware configuration can be decided. The procedure for estimating hardware requirements is explained in Chapter III, along with sample calculations for the correlator and sonobuoy graphs.

D. WALK-THROUGH FOR THE CORRELATOR GRAPH

The first step in creation of any data-flow graph is to determine the topology required based on the primitives available and the function to be performed. The correlator graph topology is given in Figure 2.6 below.

The topmost structure is a family of input queues. All structures are implemented by choosing *introduce* from the top-level pop-up menu, then selecting the structure and finally the kind, *single* or *family*. The *other* choice in any menu will return the user to the next highest menu level.

Next the various nodes are introduced, using the above procedure. Editing of the nodes may occur at any time. The full graph topology may be entered first, then edited, or each structure may be edited as it is introduced. Editing is accomplished by selecting the node with the cursor arrow and left mouse button, then selecting *edit node* and view. This creates an editing dialogue box with an iconic representation of the node with the input and output queues labelled with their associated queues. Order of the input and output ports on the icon is **the** order in which data is expected at the node. This is critical when using the input and output macros since their numbers must correspond to the relative order in which they appear on the iconic representation.

Using the *Band1* node and its succeeding local queue as an example, the node is first selected using the mouse, as described in the ECOS tutorial [Ref. 9:p. 17-20]. Next, *%%EDIT* node is selected, followed by *view*. This displays a window with an icon view of the node and its connections, as well as a NEP text area with *NOP* selected as the PRIM.

PRIM is a mandatory items to be included for all nodes. It indicates the PID to be linked to that node, and for the *Band1* node is CDM_RVF.

Next, the PRIM_IN quantities, which are the data ports, are initialized. Here, they are Lscan, Unused, 1024, F, SR, $IN0 and $IN1. The first five are parameters required by the node for setting up processing of the data. Unused indicates an optional input. Formal GIPs and literals are also provided to the node. The macro $IN0 indicates the data to be processed is on the first data channel, and carries the feedback data. The second macro, $IN1 indicates the actual data from the preceding queue. The items must be ordered exactly as they appear in the PID. One method to verify this is to invoke *primr* from another window. The alternative is to refer to the Primitives library [Ref. 13].

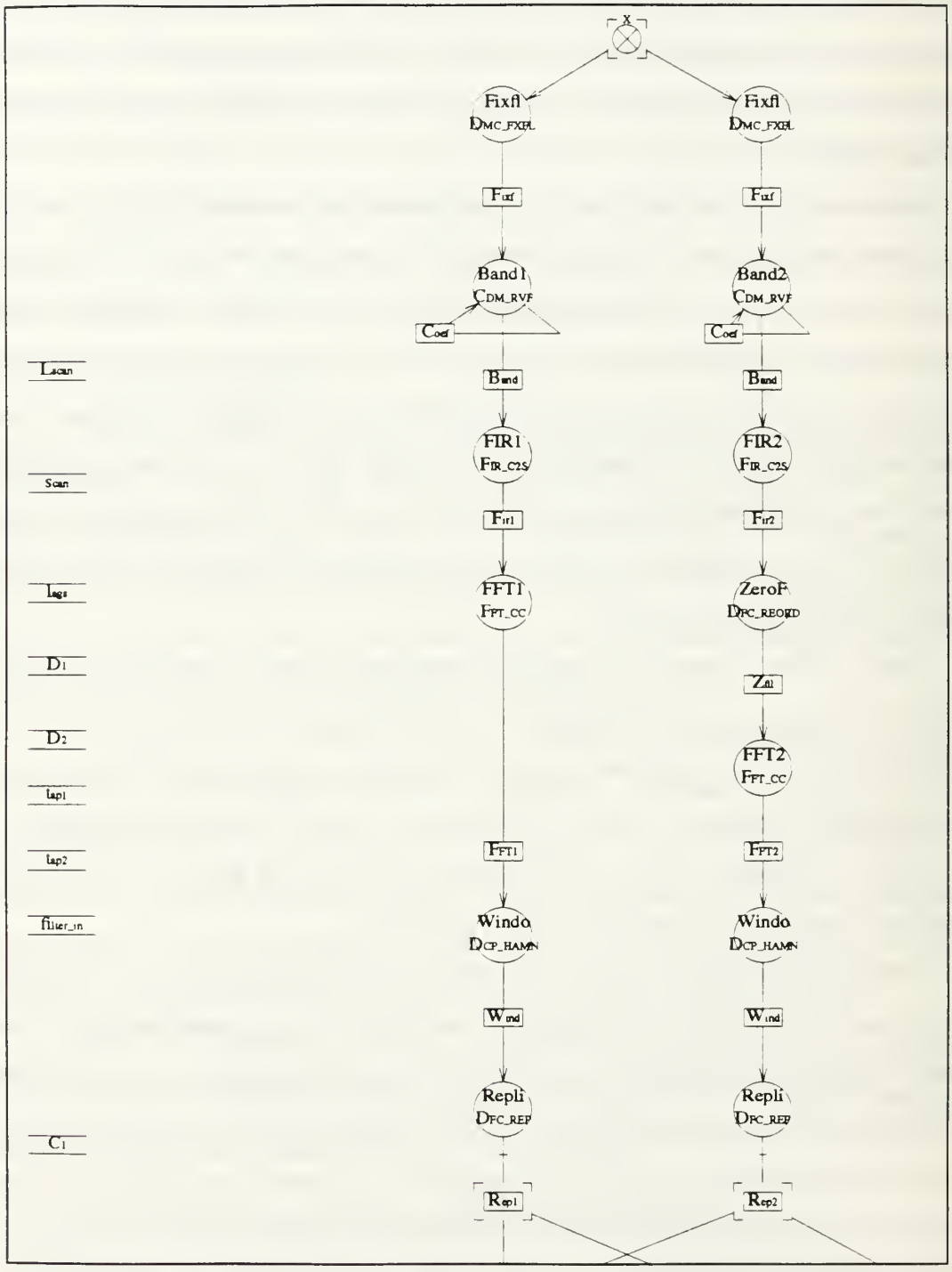


Figure 2.6: Gred Representation of the Correlator Graph (Part 1).

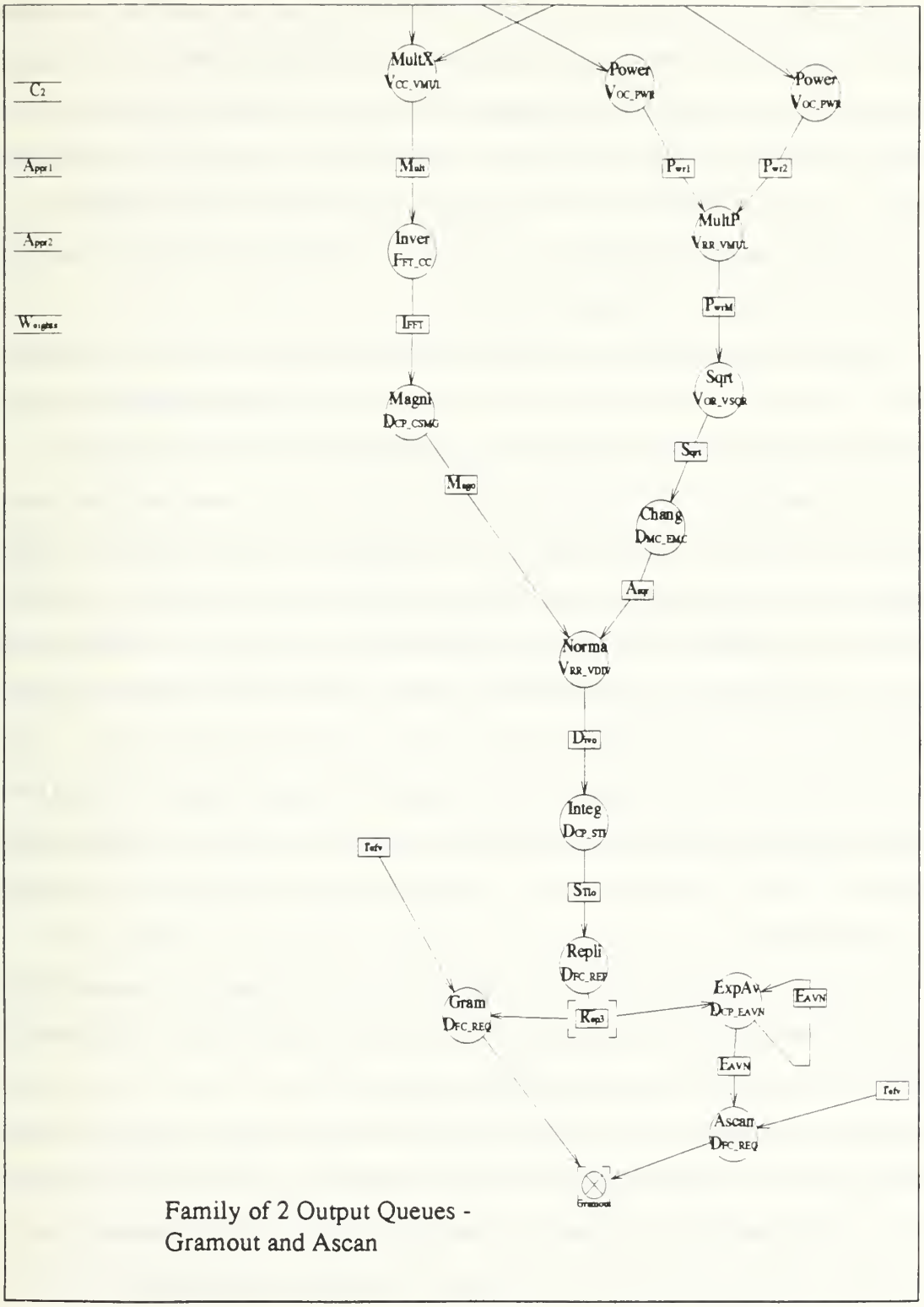


Figure 2.6: Gred Representation of the Correlator Graph (Part 2).

The other mandatory item is the PRIM_OUT. This parameter specifies the output channel to which data from the execution routine is to be placed. Again, order is critical here, since several channels may be present for data from various portions of the node routine. Notice that the output macro $OUT1 corresponds to the input macro $IN0, in the *Band1* node, and the $OUT0 macro is associated with the feedback queue and $IN1.

Other NEP items include PIP_IN and PIP_OUT, which indicate that data streams not specified in the PID are to be used by the node.

The other item which must be initialized is the local index, if the node is of kind family. In the case of a node, the local index is used by the associated ports and queues to differentiate family members. No local indices are required for the correlator graph.

Local queues must now be added to buffer the data between node executions. Local queues are introduced in a like manner to nodes. Editing initializes the mode, family dimension, local index and initial values fields. For the *Band1out* queue, the mode is *DCFLOAT*, for double precision complex float. This queue is also initialized to a 39 element vector, each of value 0.0,0.0. The procedures and limitations cited in the User's manual [Ref. 7:App. A] must be followed precisely to ensure error-free data flow.

As ports are introduced, the source and sink structures must be specified. Output ports control the produce amounts placed on the local queue. If left unedited, the default is to the full amount produced during the current execution of the node, as is the case for the *Band1* to *Band1out* port.

If data is to be written to the local queue prior to completion of node execution, a produce amount should be specified. Under most circumstances this is left unedited (blank), as is the case for all correlator graph output queues.

Feedback queues, as seen on the *Band1* and *Band2* nodes of Figure 2.6, are automatically generated when the source and sink for a port are the same node. Editing is, naturally, still required, but is identical in procedure to all other nodes and connections.

Three replicate nodes are used in the correlator graph to create multiple copies of a single data stream. These nodes are assigned zero execution time since they are often

realized as a GM function, rather than an AP function. Use of replicate nodes usually results in savings in data transfer, scheduling and execution times [Ref. 11:p. 3.19-3.21].

Translation of the graph into SPGN is accomplished with the *grail* function using a command line statement of the form *grail -g gname sname*. The flag *-g* directs the function to order GIPs so they will be declared properly [Ref. 7:p. 4.2]. File names for the graph, *gname*, and the SPGN source, *sname*, must also be specified. For the correlator graph, saved as *corr.g*, the translation command is *grail -g corr.g corr.src*. The SPGN produced is included as Appendix A. Other flags are available to create attribute tables or produce a hard copy of the graph topology.

To compile the SPGN, a command program must be written to provide the necessary formal parameters and define the I/O interfaces. Appendix B contains the command program written for the correlator graph. A declarations section is used to declare variables, the I/O procedures along with their input and output queues, and the associated graph. Four formal GIPs are used in the sample command program; *STI*, *F*, *SR*, and *TC*. Four I/O procedures along with the two input queues and two output queues are declared, as is the graph.

Declared queues and graph variables must be created using the *%CREATEQ* and *%CREATEGV* commands respectively. The four queues in the sample command program are *ib* and *ob*, each with two family members.

The remainder of the program instantiates the declared items, initializes and starts the I/O procedures and prepares the compiled file to generate an executable *ets++* input file.

Compilation of the SPGN and command program are most efficiently accomplished using the *ggcc* macro. This executes the four compilation steps which the *ggcc* acronym comprises; *grasp*, *glitr* and *cpcc* for the SPGN source and *cpcc* for the command program. Command line execution has the form *ggcc sname cpname*, with the SPGN source file name and command program name designated as *sname* and *cpname* respectively.

At the completion of compilation, syntactical and mode matching checks have been conducted and connectivity has been validated. Generation of the *ets++* input file will

extract the appropriate information from the PIDs and check the expected PID input modes and values against those indicated in the compiled graph file. The checks are listed in the *%TEST* section of each PID. Successful completion of this step results in an *ets++* input file which may be statically analyzed using *gas* or dynamically emulated using *ets++*.

III. CHARACTERIZATION OF SIGNAL PROCESSING GRAPHS

EWS provides a number of tools for determining the pertinent parameters of a signal processing graph. The static graph analyzer, *gas*, provides application-specific data which aid in determining hardware requirements. The Event Time Simulator, *ets++*, provides overhead values for node setup and breakdown, as well as individual FE loading and efficiencies to test configuration decisions made on *gas* results.

In this chapter the correlator and sonobuoy graphs are used to demonstrate the utility of the EWS suite. Emphasis will be placed on those characteristics most affected by revolving cylinder restructuring.

For applications using more than one graph, the command program specifies the interaction between the graphs within the application. This permits control of specified parameters without requiring recompilation.

Gas provides four types of graph analysis; execution statistics, connectivity, node scheduling consistency checks, and deadlock analysis [Ref. 7:p. 9.2-9.4]. Execution statistics for total node scheduling rate, total AP/AU cycles required for graph execution and System Language (SL) message rates are provided. Also data transfer rates and maximum bandwidth in 16-bit words for AP/GM and IOP/GM transfers, and total GM bandwidth.

Minimum required number of processors can be found for a given processor speed by Equation 3.1. Since *gas* performs the analysis using specified data rates, conservative choices for data rates in the I/O file will ensure minimum processing configuration needed to meet expected system demands.

$$\text{minimum processors} = \frac{\text{total required AU cycles}}{\text{processor speed}} \qquad (3.1)$$

Gas can also provide minimum required number of Global Memory modules for an application by means of the *Bandwidth GM* statistic. Transfer of data over the DTN by a

GM occurs at 20 million 16-bit words per second [Ref. 1:p. 2.15]. Therefore, minimum GM requirement is given by Equation 3.2.

$$\text{Required GM} = \frac{\text{Bandwidth GM}}{20{,}000{,}000} \tag{3.2}$$

I/O processors are seldom overloaded in EMSP or EWS applications, provided the 15 channel limit is not exceeded. AP, GM and IOP numbers are the only user definable FEs in EWS.

Other interesting statistics are the *mean AIS size* and *I/O rate (words/sec.)*. Comparison of these two quantities gives an indication of the amount of node setup required (*mean AIS size*) in contrast to the amount of data processed on the average for each node (*I/O rate*). The smaller the ratio of setup to actual execution time, the less likely the AP will experience dead time between the completion of breakdown for one node and the completion of setup of another. This is seen Figure 3.1 below.

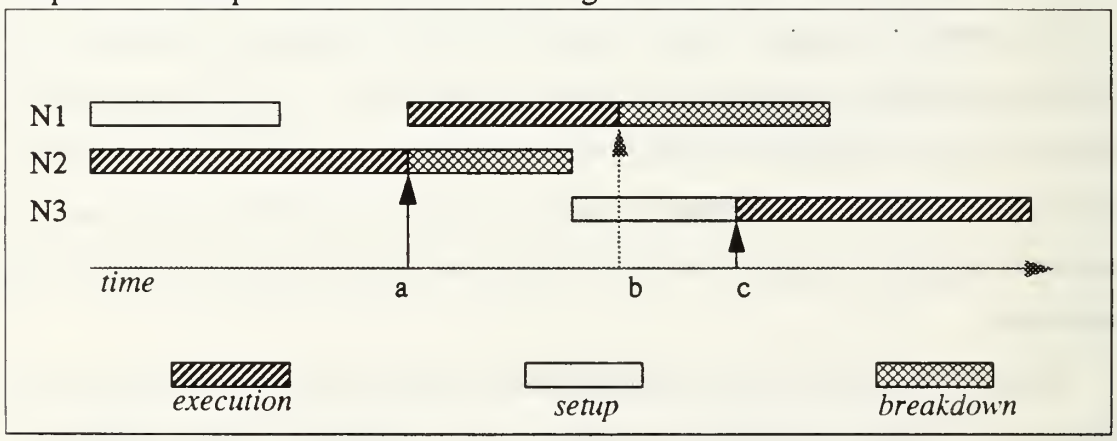


Figure 3.1: Setup, Breakdown and Execution Relationships in EMSP

Node *N1* is setting up while node *N2* is being executed. Since the setup phase for *N1* completes prior to the completion of execution for *N2*, execution of *N1* may begin almost immediately, as indicated by the arrow at *a*. The breakdown of *N2* starts and completes prior to the completion of *N1* execution, at which time the setup for *N3* may commence. Execution of *N1* completes prior to *N3* setup, shown by the arrow at *b*, and so the Arithmetic Unit will be idle until setup for *N3* completes at time *c*.

A. CORRELATOR GRAPH DATA

Figure 2.6 depicts a simple correlator graph topology consisting of 27 nodes and 37 queues. The associated SPGN is supplied as Appendix A. Two input queues (represented by the family input queue construct) feed the graph, and two output queues provide the results to the IOP. These external connections are specified in the command program, along with formal Graph Instantiation Parameters (GIP) and Graph Variables (GV). Four formal GIP, are passed to the correlator graph from the command program (Appendix B).

Another external file specifies the data rates, type of I/O procedure (Input, Output or Bidirectional), number of ports servicing the procedure and other details. The correlator graph I/O file is included as Appendix C.

To demonstrate the use of *gas* to estimate hardware requirements, two arbitrary data rates are chosen. Using the *gas* outputs for the correlator graph seen in Figures 3.2 and 3.3 below, produced using input data rates of 20,480 16-bit words per second (wps), and 2,048,000 wps on both input queues, the following parameters are derived using Equation 3.1. .

TABLE 3.1: CORRELATOR GRAPH PARAMETERS

Graph Parameter	Data Rate	Min. # of Processors
Minimum Required Processors	20480	1
Minimum Required GM	20480	1
Minimum Required Processors	2048000	6
Minimum Required GM	2048000	6

Gas output may also be used to determine maximum throughput for a given configuration. For example, if 4 APs are available, or 28 million cycles per second (SEM B), and *gas* output for a certain data rate, n, indicates 4 million cycles per second are required, then the maximum data rate supportable should be 28 million divided by 4 million or 7 times n words per second. This is generalized in Equation 3.3 below.

```
Static Graph Analysis - EWS Release: 5.6

Invocation: gas -g cbase.ets -i 10.io -n all -q all -o cbase.gout

total graph objects: 27 nodes, 37 queues, 0 graphvars, 4 I/O procedures

graph: E006
pids: /home3/swank/Ecosws/Pids/pidtoc
graph objects: 27 nodes[0,26], 37 queues[0,36], 0 graphvars, 4 I/O procedures[0,3]

graph source: cbase.ets
i/o procedure source: 10.io

 total node scheduling rate:     47.50 / second
 AP node scheduling rate:        45.00 / second
 total required AU cycles:    4.15e+05 cycles / second
 mean AU cycles per node:     9.22e+03 cycles
 mean queue consumes / node:      1.31 CQ    / node
 mean queue consume amount:    9522.81 words / CQ
 mean queue reads / node:         1.31 RQ    / node
 mean queue read amount:       9573.11 words / RQ
 mean queue writes / node:        1.17 WQ    / node
 mean queue write amount:      9900.71 words / WQ
 mean graph variable reads:       0.00 RGV   / node
 mean gv read amount:             0.00 words / RGV
 mean graph variable writes:      0.00 WGV   / node
 mean gv write amount:            0.00 words / WGV
 mean queue+gv read amount:    9573.11 words / (RQ,RGV)
 mean queue+gv write amount:   9900.71 words / (WQ,WGV)
 mean AIS size:              256.00 words / AIS
 Bandwidth AP:               1.08e+06 words / second
 Bandwidth IOP:              4.22e+04 words / second
 Bandwidth GM:               1.12e+06 words / second
```

Figure 3.2: Gas Output for the Correlator Graph at 20480 words per second

Using Equation 3.3, maximum throughput was calculated for both *gas* analyses, using four processors as a sample configuration. The results are tabulated in Table 3.2.

$$\text{Max\_throughput} = \frac{\text{No\_processors} \times 7\text{MHz}}{\text{total\_required\_AU\_cycles}} \times \text{GAS\_data\_rate} \qquad (3.3)$$

TABLE 3.2: MAXIMUM THROUGHPUT FOR CORRELATOR GRAPH

GAS Data Rate (wps)	Max Throughput (wps)
20,480	1,381,783
2,048,000	1,381,783

```
Static Graph Analysis - EWS Release: 5.6

Invocation: gas -g cbase.ets -i 1000.io -n all -o cbase1000.gout

total graph objects: 27 nodes, 37 queues, 0 graphvars, 4 I/O procedures

graph: E006
pids: /home3/swank/Ecosws/Pids/pidtoc
graph objects: 27 nodes[0,26], 37 queues[0,36], 0 graphvars, 4 I/O procedures[0,3]

graph source: cbase.ets
i/o procedure source: 1000.io

 total node scheduling rate:   4750.00 / second
 AP node scheduling rate:      4500.00 / second
 total required AU cycles:    4.15e+07 cycles / second
 mean AU cycles per node:     9.22e+03 cycles
 mean queue consumes / node:      1.31 CQ   / node
 mean queue consume amount:    9522.81 words / CQ
 mean queue reads / node:         1.31 RQ   / node
 mean queue read amount:       9573.11 words / RQ
 mean queue writes / node:        1.17 WQ   / node
 mean queue write amount:      9900.71 words / WQ
 mean graph variable reads:       0.00 RGV  / node
 mean gv read amount:             0.00 words / RGV
 mean graph variable writes:      0.00 WGV  / node
 mean gv write amount:            0.00 words / WGV
 mean queue+gv read amount:    9573.11 words / (RQ,RGV)
 mean queue+gv write amount:   9900.71 words / (WQ,WGV)
 mean AIS size:              256.00 words / AIS
 Bandwidth AP:              1.08e+08 words / second
 Bandwidth IOP:             4.22e+06 words / second
 Bandwidth GM:              1.12e+08 words / second
```

Figure 3.3: Gas Output for the Correlator Graph at 2048000 Words Per Second.

The *gas* output files for the initial data rate of 20480 words per second is given in Appendix C with all optional data included. One of the options provides individual node statistics, including node execution rate, expressed as executions per second and cycles per second. The execution time for each node may be extracted from these figures.

Execution time is only part of the total cycles required to dispatch a node. As previously discussed, once a node is matched to a processor by the scheduler, instructions must be fetched and data loaded into local memory. This is the setup phase. Once the execution of the node is complete, the data derived during execution is written to an assigned queue during the breakdown phase.

The comparative sizes of setup, breakdown and execution time can have a significant impact on graph execution efficiency. From the *ets++* emulation for the correlator at 20480

wps data rate, and using a four processor configuration, setup and breakdown times were derived. Combining these with the *gas* analysis execution times, a complete characterization of individual node contributions to graph execution may be made. This is shown in both tabular and graphical form below as Table 3.3 and Figure 3.4. All times are in milliseconds.

TABLE 3.3: CORRELATOR GRAPH NODE PARAMETERS

No.	Node	Exec	Set up	Brk dwn	No.	Node	Exec	Set up	Brk dwn
1	FIXFL1	0.9	2.3	2.8	15	POWERX	0.4	4.3	0.8
2	FIXFL2	0.9	2.3	2.9	16	POWERY	3.4	7.6	1.5
3	BAND1	2.4	4.0	5.5	17	IFFT	0.3	2.9	2.3
4	FIR1	9.7	9.7	1.7	18	MAGNI	<0.1	1.6	0.7
5	BAND2	2.4	5.4	8.7	19	MULTPWR	<0.1	1.8	0.5
6	FIR2	9.7	7.1	1.7	20	SQRT	0	0.8	3.8
7	ZEROFILL	1.2	2.3	1.7	21	CHANGE	0.7	0	2.2
8	FFT2	3.1	9.8	1.6	22	NORMAL	1.0	1.0	0.5
9	FFT1	3.1	2.4	1.6	23	INTEGR	1.0	1.3	0.5
10	WINDOW1	0.7	2.4	1.9	24	REPLIC3	0	0	1.7
11	WINDOW2	0.7	8.7	1.6	25	GRAM	1.3	1.2	0.7
12	REPLIC2	0	0	7.2	26	EXPAVG	0.5	1.2	0.8
13	REPLIC1	0	0	1.9	27	ASCAN	1.3	3.2	1.5
14	MULTXY	1.0	3.8	1.8					

Execution times calculated from *gas* may be used for any data rate or machine configuration providing the NEP parameters (threshold, read, consume and produce) remain unchanged.

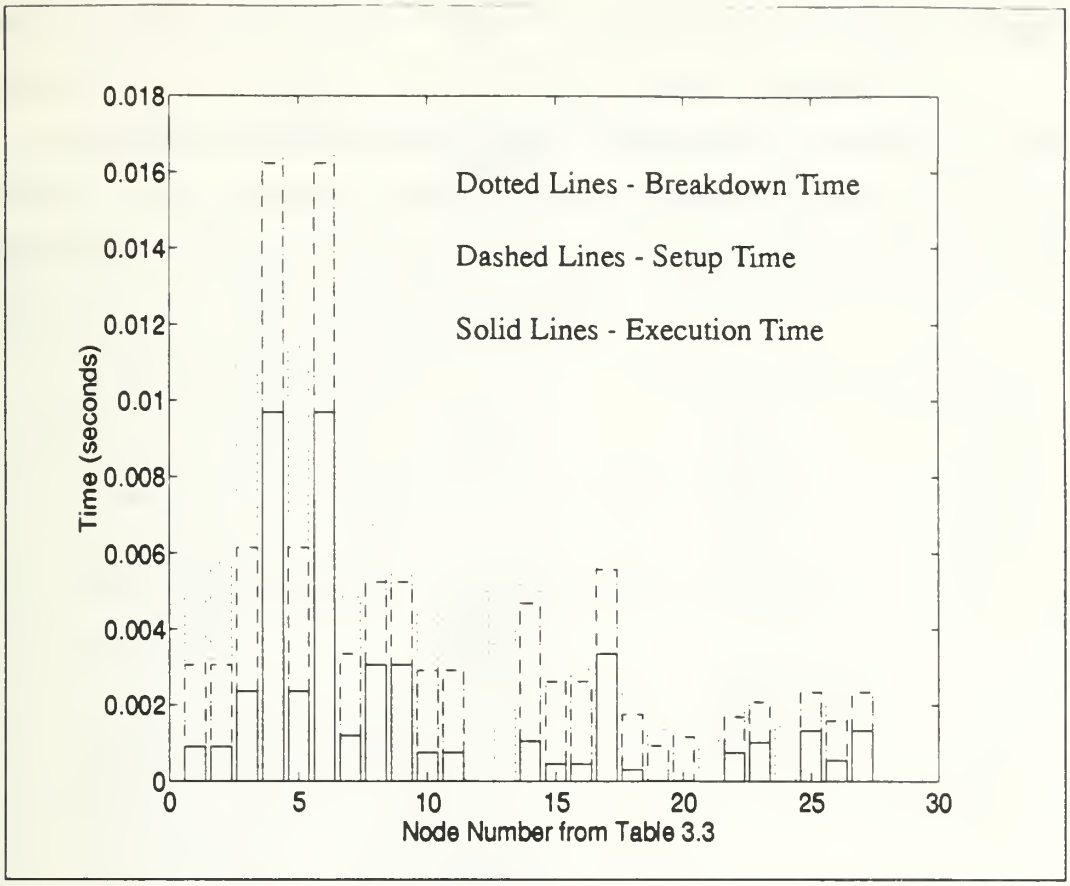


Figure 3.4: Setup, Breakdown and Execution times for the Correlator Graph

B. ACTIVE SONOBUOY GRAPH CHARACTERIZATION

A chained cluster of nodes is treated, in all respects, as a node by the scheduler, APs, GMs and buses. Data provided, therefore, cannot reflect execution, setup and breakdown for individual nodes within a chain.

The root level active sonobuoy graph is reproduced below as Figure 3.5. Subgraph topologies may be found in Appendix E, with the associated SPGN in Appendix F.

Another aspect of using multiple graphs to construct an end application is the ability to provide as many instances of each as is required by the current situation. Changes in the tactical situation, could, for example, require increased active sonobuoy channels and reduced passive capability. This can be accomplished in EMSP completely by use of the command program, and simulated in EWS by use of the I/O file and invocation of multiple

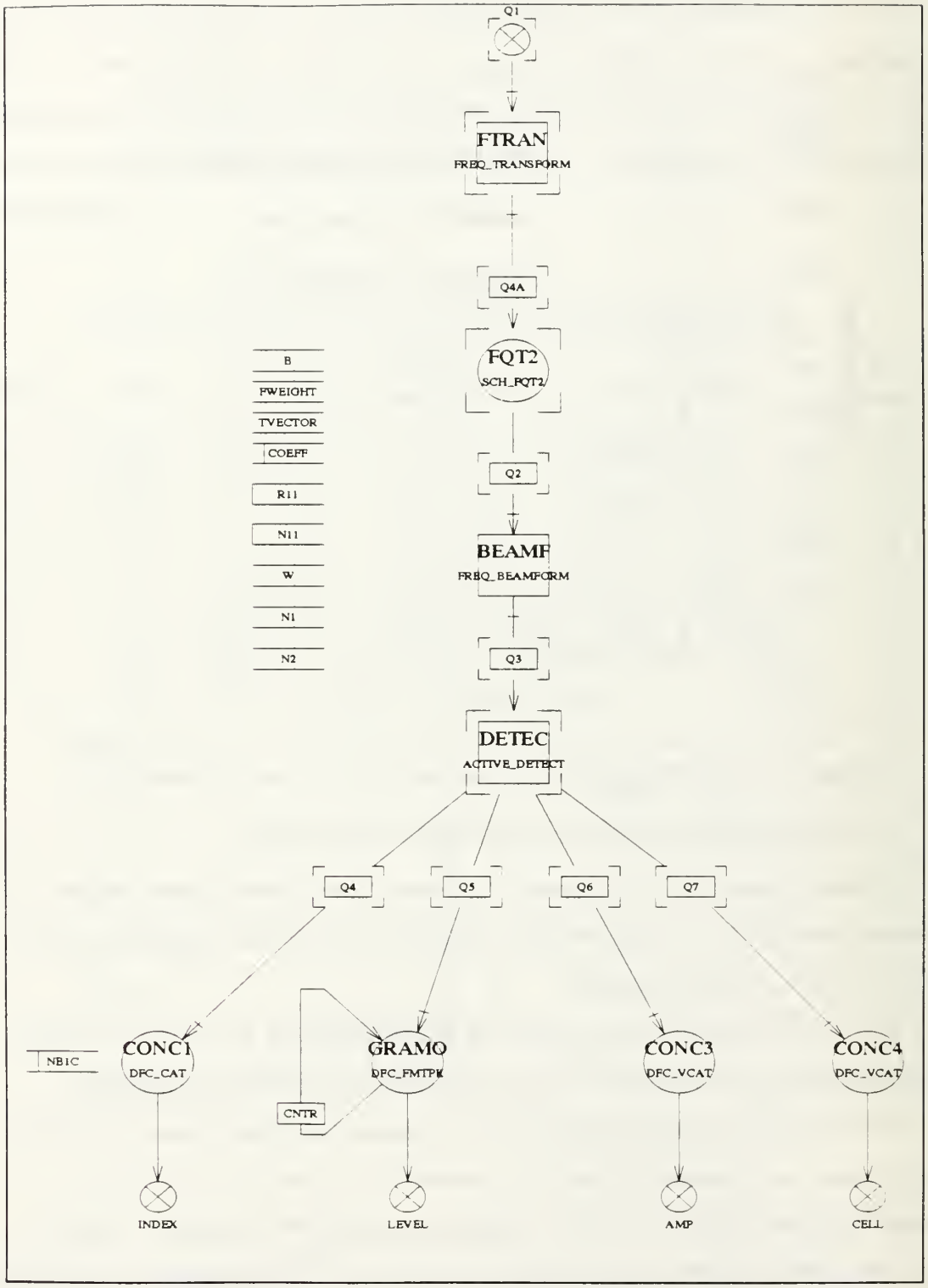


Figure 3.5: Root Level Active Sonobuoy Graph

instances during simulation [Ref. 14]. The *ets++* command program used for benchmark testing is shown in Appendix K, and the I/O file is provided as Appendix G.

Gas analysis of the Active Sonobuoy graph was performed with a data input rate of 677 16-bit words per second on each of the 16 input channels. Results are given below in Figure 3.6. The data rate of 677 words per second was provided in the original files received from AT&T, and is apparently based on field data.

```
Static Graph Analysis - EWS Release: 5.6

Invocation: gas -g rbase.ets -i bact.io -n all -o rbase.gout

total graph objects: 41 nodes, 213 queues, 26 graphvars, 5 I/O procedures

graph: ACT_PROC
pids: /home3/swank/Ecosws/Pids/pidtoc
graph objects: 41 nodes[0,40], 213 queues[0,212], 26 graphvars[0,25], 5 I/O procedures[0,4]

graph source: rbase.ets
i/o procedure source: bact.io

 total node scheduling rate:      8.56 / second
 AP node scheduling rate:         7.90 / second
 total required AU cycles:    6.87e+05 cycles / second
 mean AU cycles per node:     8.70e+04 cycles
 mean queue consumes / node:      6.69 CQ   / node
 mean queue consume amount:     874.13 words / CQ
 mean queue reads / node:         6.69 RQ   / node
 mean queue read amount:        963.45 words / RQ
 mean queue writes / node:        6.44 WQ   / node
 mean queue write amount:       859.37 words / WQ
 mean graph variable reads:       4.08 RGV  / node
 mean gv read amount:         10701.43 words / RGV
 mean graph variable writes:      0.00 WGV  / node
 mean gv write amount:            0.00 words / WGV
 mean queue+gv read amount:    4655.22 words / (RQ,RGV)
 mean queue+gv write amount:    859.37 words / (WQ,WGV)
 mean AIS size:              1194.67 words / AIS
 Bandwidth AP:                4.4e+05 words / second
 Bandwidth IOP:               8.31e+03 words / second
 Bandwidth GM:                4.48e+05 words / second
```

Figure 3.6: Gas Analysis of the Active Sonobuoy Graph at 677 Words Per Second on Each Channel.

A comparison of mean AIS size to mean queue consume amount indicates an overall execution to overhead ratio of approximately 1.37:1. Individual node parameters in milliseconds are tabulated in Table 3.4 and graphically depicted in Figure 3.7. While the Active Sonobuoy and Correlator graphs are quite different in functionality, the advantages

of chaining are suggested by the increased execution to overhead ratios for chained entities such as FQT1 in the Active Sonobuoy graph.

TABLE 3.4: ACTIVE SONOBUOY GRAPH NODE PARAMETERS

No.	Node	Exec	Set up	Brk dwn	No.	Node	Exec	Set up	Brk dwn
1	CONCAT4	0.60	3.82	3.80	22	[1]DETECT	8.83	1.58	1.19
2	CONCAT3	0.60	3.82	3.54	23	[2]DETECT	8.83	5.38	1.19
3	GRAM	2.38	3.99	3.37	24	[3]DETECT	8.83	9.02	1.19
4	CONCAT1	0.16	3.63	5.33	25	[4]DETECT	8.83	9.02	1.19
5	[1]FQT2	12.6	13.0	1.29	26	[5]DETECT	8.83	9.02	1.19
6	[2]FQT2	12.6	13.2	1.29	27	[6]DETECT	8.83	9.02	1.19
7	[3]FQT2	12.6	12.7	1.29	28	[7]DETECT	8.83	9.02	1.19
8	[4]FQT2	12.6	12.7	1.29	29	[8]DETECT	8.83	9.02	1.19
9	[5]FQT2	12.6	12.7	1.29	30	[9]DETECT	8.83	9.02	1.19
10	[6]FQT2	12.6	12.7	1.29	31	[10]DETEC	8.83	9.02	1.19
11	[7]FQT2	12.6	12.7	1.29	32	[11]DETEC	8.83	9.02	1.19
12	[8]FQT2	12.6	12.7	1.29	33	[12]DETEC	8.83	9.02	1.19
13	[9]FQT2	12.6	12.7	1.29	34	[13]DETEC	8.83	9.02	0.81
14	[10]FQT2	12.6	12.7	1.29	35	[14]DETEC	8.83	9.02	0.81
15	[11]FQT2	12.6	12.7	1.29	36	[15]DETEC	8.83	9.02	1.01
16	[12]FQT2	12.6	12.7	0.91	37	[16]DETEC	8.83	9.02	0.81
17	[13]FQT2	12.6	12.7	1.29	38	[1]FQT1	13.3	7.66	6.80
18	[14]FQT2	12.6	12.7	0.91	39	[2]FQT1	13.3	7.26	6.70
19	[15]FQT2	12.6	12.7	0.91	40	[3]FQT1	13.3	13.5	6.80
20	[16]FQT2	12.6	12.7	0.91	41	[4]FQT2	13.3	13.5	6.80
21	BEAMF	24.6	31.2	4.55					

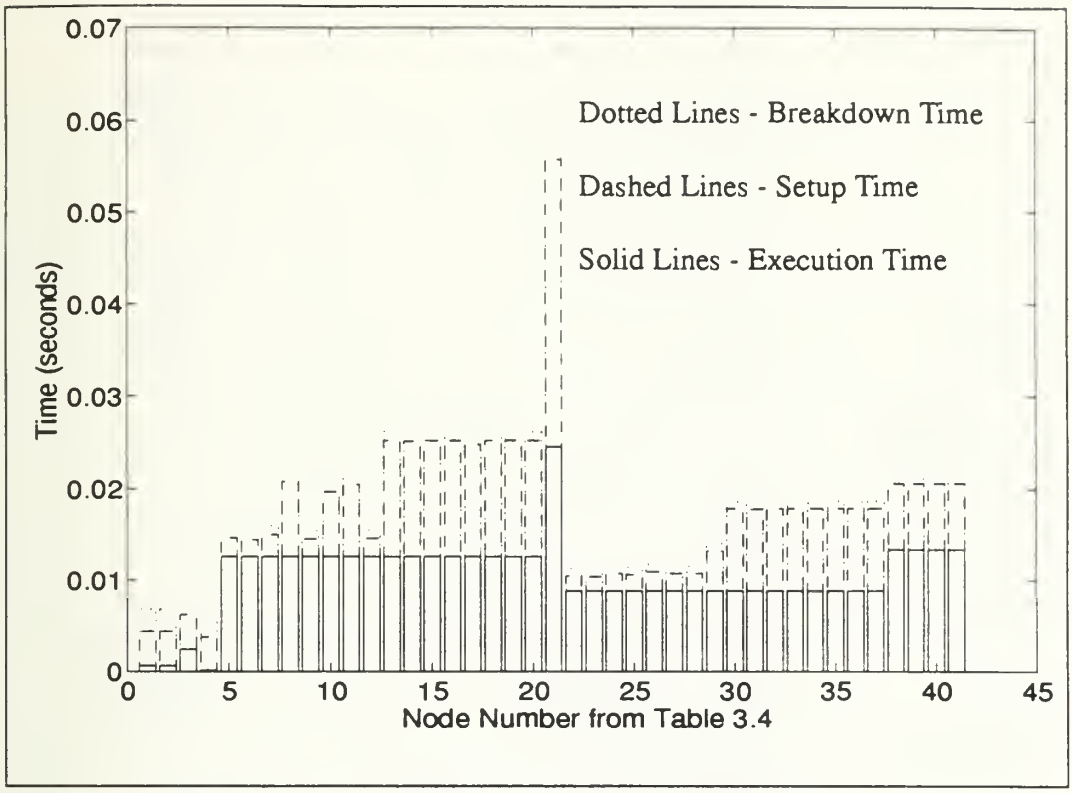


Figure 3.7: Active Sonobuoy Graph Node Parameters

C. CHARACTERIZATION OF THE PASSIVE SONOBUOY GRAPH.

The Passive Sonobuoy graph actually consists of two separate graphs, each incorporating at least one subgraph. The graphs are dynamically linked using the command program. Figure 3.8 is the Octave Filter root level graph, whose output queue, Q7 is linked to the Frequency Analysis graph input queue, Q1.

Figure 3.9 is the Frequency Analysis root level graph. Complete graph topologies are included as Appendices K and M for the Octave Filter and Frequency Analysis graphs respectively. Appendices L and N are the associated SPGN for the graph topologies. The command program is Appendix O, and is annotated to indicate the graph linkage portions.

A portion of the *gas* output for the passive graph is given in Figure 21 below. Section 2 of the I/O program, *bmk-all.io*, included as Appendix H, drives the graph with a 6503 word per second input data rate. Using the same eight processor configuration utilized for

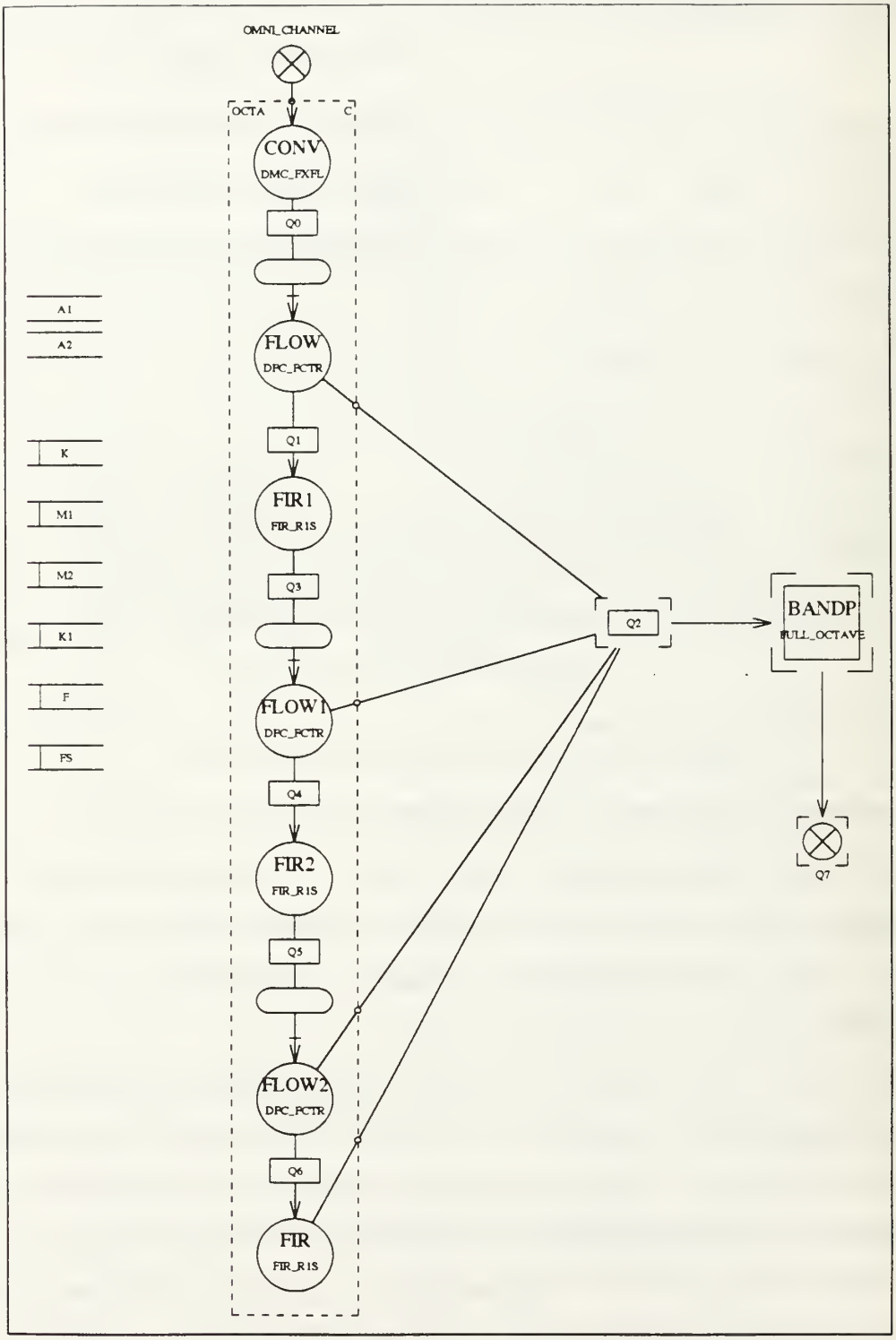


Figure 3.8: Octave Filter Portion of the Passive Sonobuoy Graph.

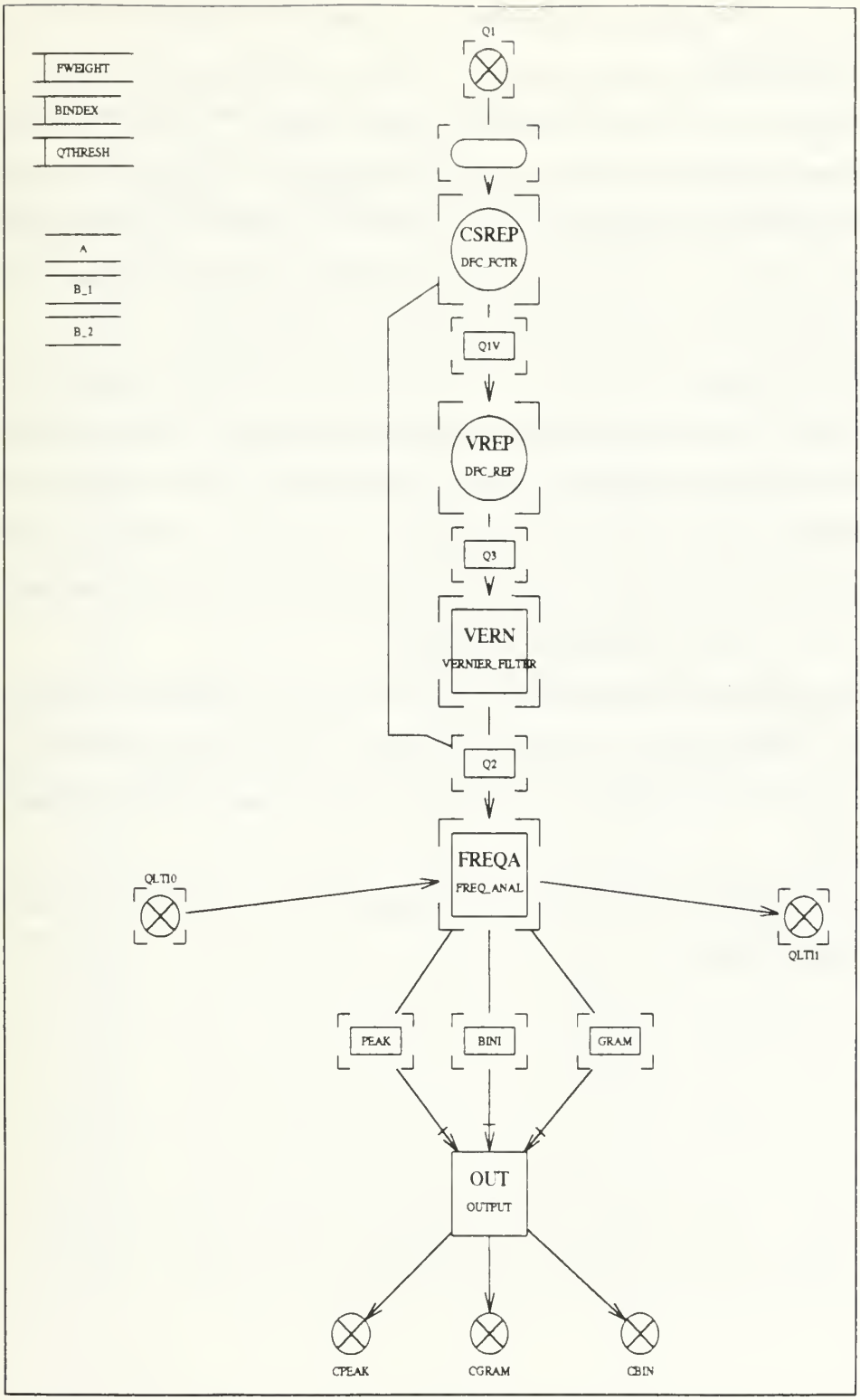


Figure 3.9: Frequency Analysis Portion of the Passive Sonobuoy Graph

the Active Sonobuoy graph, and Equation 3.3, a maximum throughput of 655,755 words per second is indicated.

Individual node statistics for execution, setup and breakdown time are again provided in tabular and graphical form below as Table 3.5 and Figure 3.10. Of note is the VREP node (number 6) of the Frequency Analysis graph. This is an example of a sophisticated vector replication node which is entirely a function of the GM. For this reason it does not appear in the *gas* output, and requires no setup or breakdown time [Ref. 12].

The I/O file used with both the Active and Passive Sonobuoy graphs, contains a section which initializes and instantiates a family of 50 instances of the passive graph. This requires a means to input the data to each of the 50 instances. Another graph, containing a single replicate node, is used for this purpose. This section has no significance for the purposes of implementing the RC algorithm, and served only to place a fixed load on the system for which the benchmarks were written.

The Frequency Analysis portion of the Passive Sonobuoy graph uses only one of the four Octave Filter outputs in the benchmark scenario. Full utilization of the Octave Filter portion would require four Frequency Analysis graphs for each invocation of the Octave Filter graph.

A warning appears with this scenario indicating that three of the Octave Filter output queues are unattached.

TABLE 3.5: PASSIVE SONOBUOY GRAPH NODE PARAMETERS

No.	Node	Exec	Set up	Brk dwn	No.	Node	Exec	Set up	Brk dwn
1	OCTAVE	5.68	1.68	1.83	22	[8]VERN	6.91	9.31	0.86
2	[1]FULL	12.6	2.48	0.86	23	[8]IIR	4.43	2.16	1.01
3	[2]FULL	12.6	2.00	0.43	24	FORMAT	2.12	2.42	1.36
4	[3]FULL	6.55	1.96	0.43	25	CONCAT2	0.31	2.22	1.35
5	[4]FULL	3.56	1.96	0.43	26	CONCAT1	0.31	2.21	1.35
6	CSREP	0	0	2.36	27	[1]TRANS	6.38	2.17	0.62
7	VREP	0	0	0	28	[1]DETECT	5.47	4.27	1.02
8	[1]VERN	6.91	3.50	0.86	29	[2]TRANS	6.38	2.48	0.62
9	[1]IIR	4.43	2.17	1.03	30	[2]DETECT	5.47	4.27	1.02
10	[2]VERN	6.91	4.28	0.86	31	[3]TRANS	6.38	2.27	0.62
11	[2]IIR	4.43	2.20	0.99	32	[3]DETECT	5.47	4.10	1.02
12	[3]VERN	6.91	5.13	0.86	33	[4]TRANS	6.38	2.25	0.62
13	[3]IIR	4.43	2.16	1.04	34	[4]DETECT	5.47	4.51	1.02
14	[4]VERN	6.91	5.98	0.86	35	[5]TRANS	6.38	2.15	0.62
15	[4]IIR	4.43	2.16	1.01	36	[5]DETECT	5.47	4.33	1.02
16	[5]VERN	6.91	6.83	0.86	37	[6]TRANS	6.38	2.16	0.62
17	[5]IIR	4.43	2.22	1.04	38	[6]DETECT	5.47	4.23	1.02
18	[6]VERN	6.91	7.68	0.86	39	[7]TRANS	6.38	2.18	0.62
19	[6]IIR	4.43	2.16	1.04	40	[7]DETECT	5.47	4.19	1.02
20	[7]VERN	6.91	8.63	0.86	41	[8]TRANS	6.38	2.26	0.62
21	[7]IIR	4.43	2.16	1.03	42	[8]DETECT	5.47	4.03	1.03

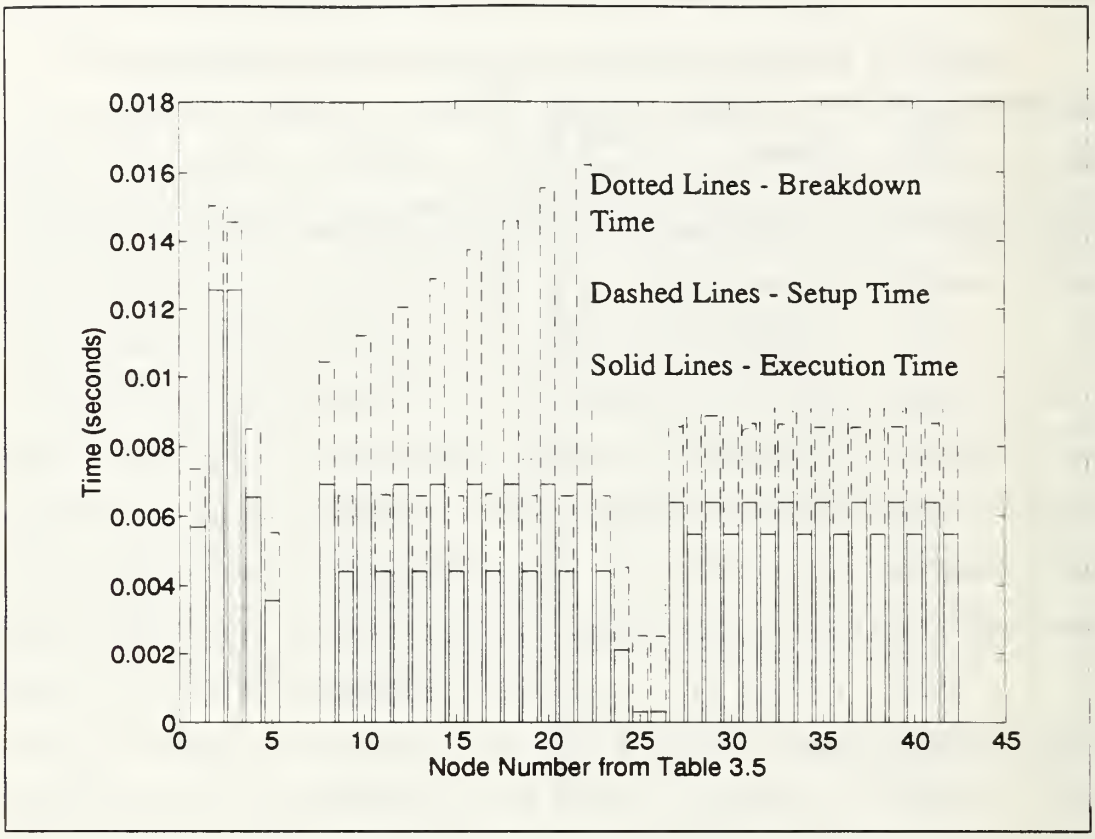


Figure 3.10: Passive Node Parameters.

IV. REVOLVING CYLINDER SCHEDULING

The Revolving Cylinder (RC) scheduling algorithm as documented in [Ref. 6] and [Ref. 15], is a packing scheme for mapping parallel program execution onto a symbolic cylinder determined by hardware resources. It is a framework for performming graph level optimization and to force run-time ordering of node execution.

The cylinder circumference is determined by dividing the total program execution time by the number of available processors. This is a first approximation and several heuristics are available for increasing this value, if needed. Length of the cylinder is determined by stacking slices of unit width, each of which represents one available processor. The final structure then has a surface area equal to, or greater than, the total execution time of the program. The following section describes the RC approach briefly.

A. THEORY

With perfect parallelism, and assuming no overhead costs, the maximum throughput possible for a data-flow graph is the total graph execution time divided by the number of available processors.

As an example, Figure 4.1 depicts a simple data-flow graph with ten nodes. Execution times are indicated beside each node and the connecting arcs indicate data flow. The machine configuration chosen will consist of four processors, identical in all respects.

The total graph execution time is 160 cycles. The maximum throughput for the graph using simple FCFS scheduling would be the sum of the nodes along the longest thread, or path, through the graph. In this case that is the thread 1-3-4-6-9, with each number corresponding to the associated graph node.

Using the Revolving Cylinder algorithm, the total graph execution time would be divided by the number of processors, four in this case, to yield a minimum cylinder circumference of 40. This is the minimum cylinder circumference since the actual node sizes may not permit optimal packing to be realized.

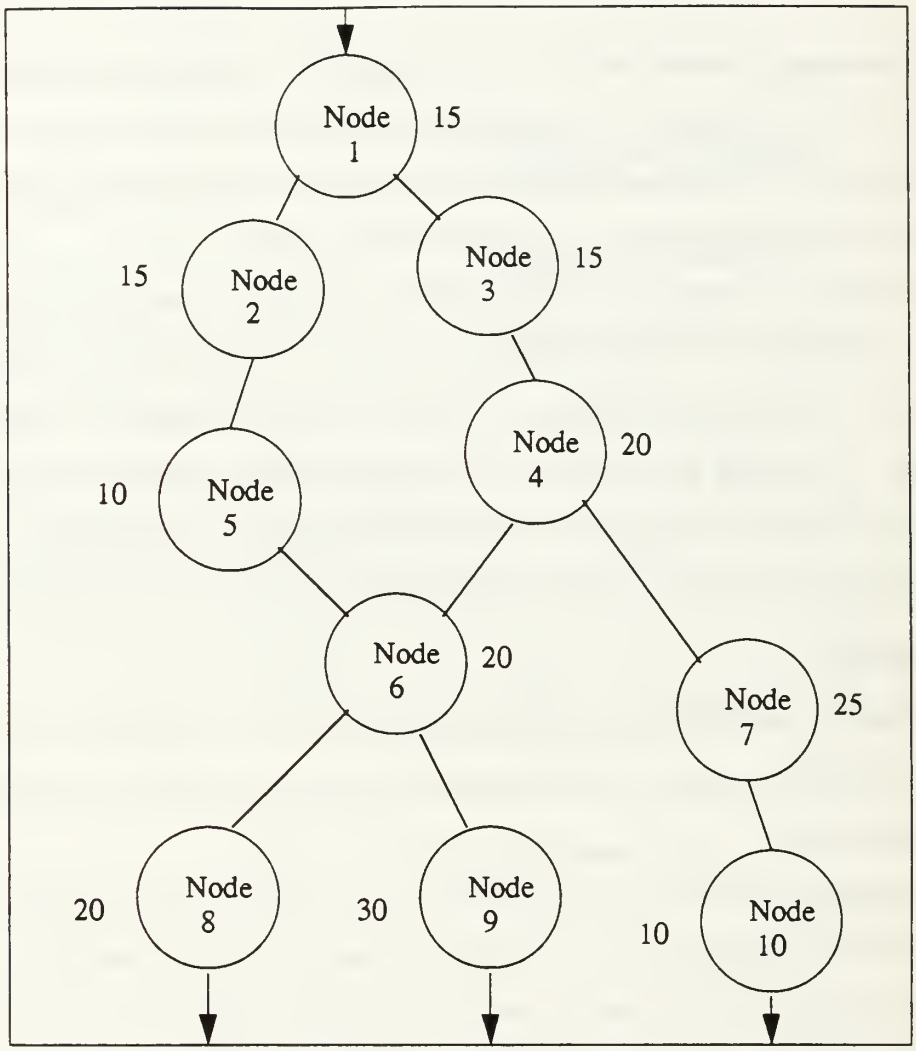


Figure 4.1: Example Node Topology

Packing of the cylinder will depend on the heuristic chosen. One choice is topological sort, where the first time segment of the first processor on the graph is filled with the first executing node of the graph, the first time segment of the second processor with the second, and so forth. This approach ensures that all processors begin processing data as soon as possible, but may result in a fragmented cylinder if large nodes are added to a processor slice late in the execution cycle. An example of this is shown in Figure 4.2 below.

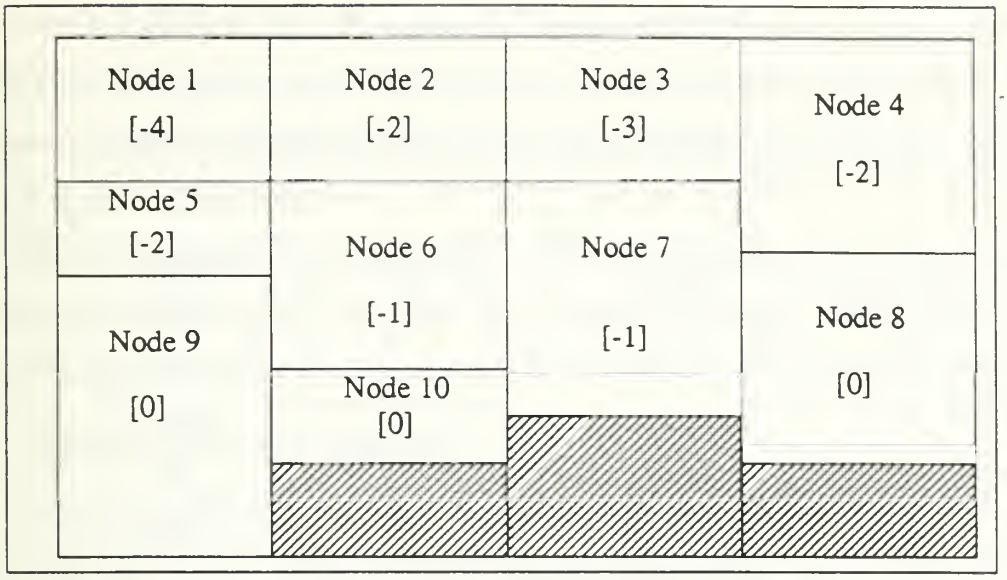


Figure 4.2: Fragmented Cyinder Packing

Cylinder circumference would obviously be greater than minimum (total execution divided by number of processors, or 40) for the above example. Using the node execution times from Figure 4.1, the circumference of the cylinder in Figure 4.2 is 55 cycles. The circumference was increased in this case only enough to fit the large node on processor one. In practice, circumference is often increased by a percentage of the minimum cylinder circumference, which would further increase the amount of unused cylinder area.

Another packing heuristic, which minimizes the fragmentation experienced with topological packing, is to sort nodes by size, then pack the cylinder using the sorted node list. This normally results in a less fragmented cylinder, but may create latency problems depending on graph topology. An example of a cylinder packed with this heuristic is shown in Figure 4.3.

Cylinder circumference established using this heuristic has been reduced to 50 cycles from 55 cycles. If input data is always available, the cylinder shown in Figure 4.3 will consume data every 50 cycles, in contrast to Figure 4.2, where input data was consumed

every 55 cycles. Throughput has increased 10 percent utilizing a different packing heuristic.

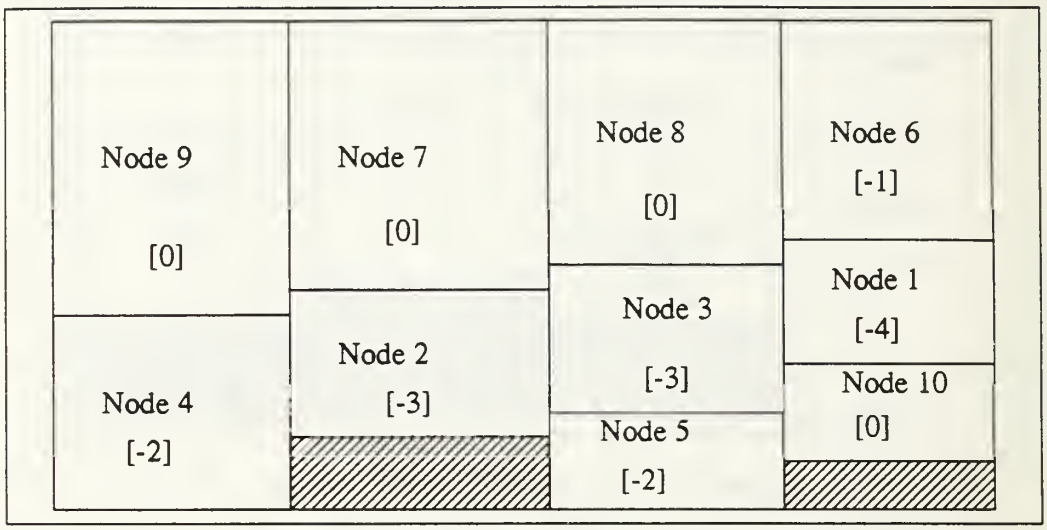


Figure 4.3: Cylinder Packed by Sorted Node List

Node 9 in this example, cannot execute before Node 1 through Node 8 of the current execution of the graph. It must therefore, be from a previous invocation of the graph. To track this, indices are assigned from 0 to -*n*, where *n* is the number of previous iterations of the graph still being processed by the system.

To assign indices, the output nodes are first assumed to be the currently executing nodes and given indices of 0. We then move backwards through the data-flow diagram and the cylinder packing to determine if a predecessor node will finish execution in time to provide its data to the node in question. If not, the data must be provided by an earlier revolution of the cylinder, and the index for the predecessor node is decremented by one. With this technique, the indices for all nodes can be assigned as shown in the bracketed numbers on Figures 4.2 and 4.3.

With the cylinder packed, and indices assigned, the task of enforcing the structure can begin. Two methods of imposing the scheduling assignments are used in the Revolving Cylinder algorithm, chaining and dependency arcs. Chaining links all, or certain, nodes

assigned to a particular processor by the packing technique. In this way the chained nodes are viewed as a single entity by the scheduler. Dependency arcs create an artificial data flow with a series of triggering pulses being produced by one node, and consumed by another. In this way, a data flow is created which can be used to force scheduling of nodes according to the cylinder assignments.

General implementation of Revolving Cylinder dependency arcs is described in detail in [Ref. 15] and simple programs to pack the cylinder and assign indices are presented in [Ref. 16]. They will be discussed in this thesis only to the degree required for clarity.

B. IMPLEMENTATION IN ECOS

In ECOS, as well as EMSP, scheduling is data-flow dependent at the macro, or graph, level. Dependency arcs must therefore consist of artificial data flows. Implementation is accomplished using the same structures seen previously for traditional data flow, node production sent to an assigned queue in memory, via an output port, then to the successor node via an input port. In the case of hierarchical structures, triggering pulses are passed using the actual input and output constructs described on page 14 of Chapter II, for the active sonobuoy graph.

Dependency arcs are not accounted for in the Primitive Interface Definitions (PID), and must be introduced using the Parallel Interface Port (PIP) [Ref. 10:p. 3.2]. The PIP allows data not specified in the PRIM_IN portion of the PID to be included in normal execution functions (i.e., consumption of data from the queues), and, more importantly, in the scheduling process. In this manner, node scheduling may be delayed until data is available on the PIP trigger queues†, as well as on the original data queues. This methodology of delaying execution of one node until completion of another is referred to as *start after finish* [Ref. 16]. It is this functionality which is exploited for RC scheduling.

---

† Trigger queue will be used in place of dependency arc for the remainder of this thesis.

1. RC Restructuring of the Correlator Graph

The packing, indexing and restructuring programs presented in Akin [Ref. 16] were used to generate the dependency arcs for the correlator graph. The inputs to the programs must be manually generated, and consist of a *graph.dat* file containing node and queue parameters, and a machine configuration file citing number of memory modules, number of I/O processors and number of Arithmetic Processors.

The mapping and restructuring programs assume equal execution rates for all graph nodes. This value is specified for each node in the *gas* output file, and is determined by the produce and consume amounts for each node. Two nodes in the correlator graph execute at one-fourth the rate of the remaining nodes; the *Change* and *Normalize* nodes. As seen in the *gas* output, the execution time of the *Change* node is 0 cycles, and 162 cycles for the *Normalize* node. With such short execution times, the mapping and restructuring was performed with these nodes bypassed.

The input files for the correlator graph, along with selected output files, are shown in Appendix L. Note the inclusion of two additional nodes in the files. These are input and output nodes, which are required for program execution, but are not part of the actual cylinder mapping.

The cylinder mapping resulting from the mapping and restructuring programs is shown in Figure 4.4 below. Node lengths within the cylinder are approximate.

Figure 4.5 is the RC restructured correlator graph. Appendix M is the SPGN associated with this restructured graph. Due to the flat, or non-hierarchical, nature of the graph, introduction of the trigger queues was straightforward. Using either the *tokens* file, or the modified *graph.dat* file, dependency arcs are introduced using *gred.*

First the trigger queues are placed using the *local queue* constructs. Families of queues are used where several nodes are dependent on the execution of a single node. The trigger queue is then connected to the appropriate nodes using either single or family ports. Family port structures are used to provide triggering *tokens* to each member of a family queue construct, or to specify the queue family member providing data to a particular node.

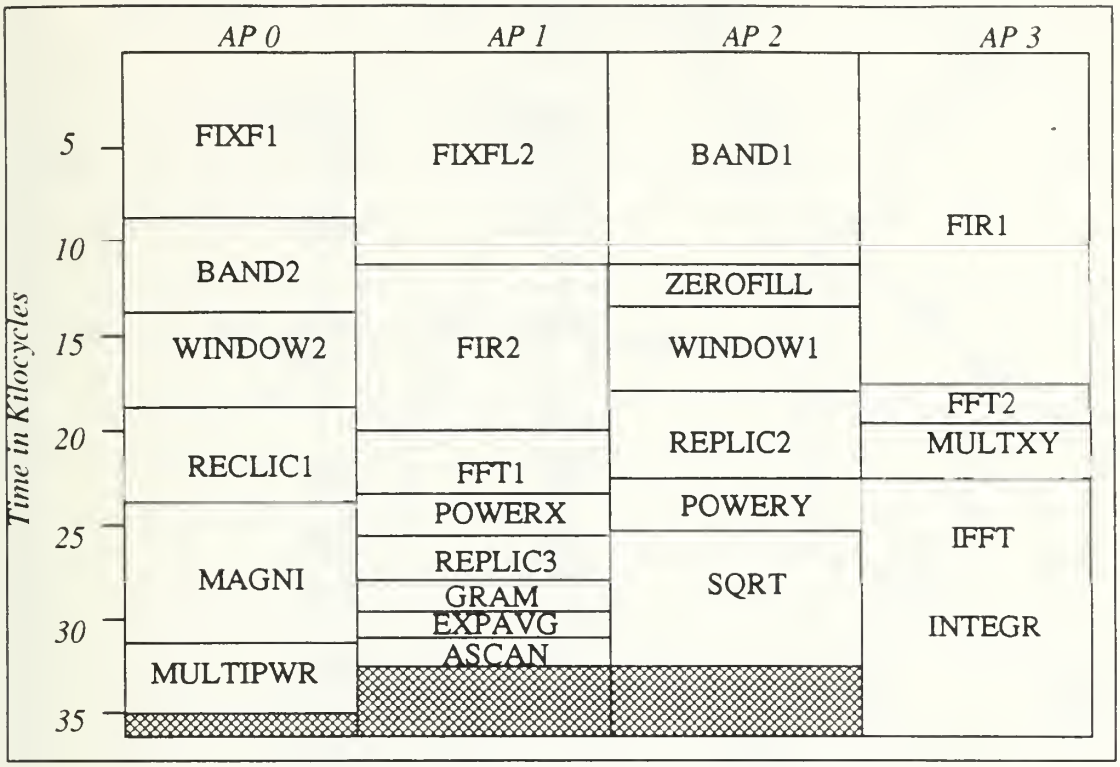


Figure 4.4: Correlator Graph Cylinder Mapping

The last step in introducing the trigger queue is editing of the queue, input and output ports, and associated nodes.

Nodes are edited to include either a PIP_IN (Parallel Interface Port-Input) if the node provides the sink for a trigger queue, or PIP_OUT (Parallel Interface Port-Output) if the node produces triggering tokens. Macros are normally used to indicate data inputs ($INx) or data outputs ($OUTx) where *x* is an integer starting at zero. Care must be taken to number the macros as they appear on the node schematic displayed when editing. Confusion here may result in *mode mismatch* or *unexpected data mode* messages during compilation.

Output ports must be initialized with the Node Execution Parameter (NEP) *pulse*, and the number of triggering tokens to be produced. Normally one token is produced for

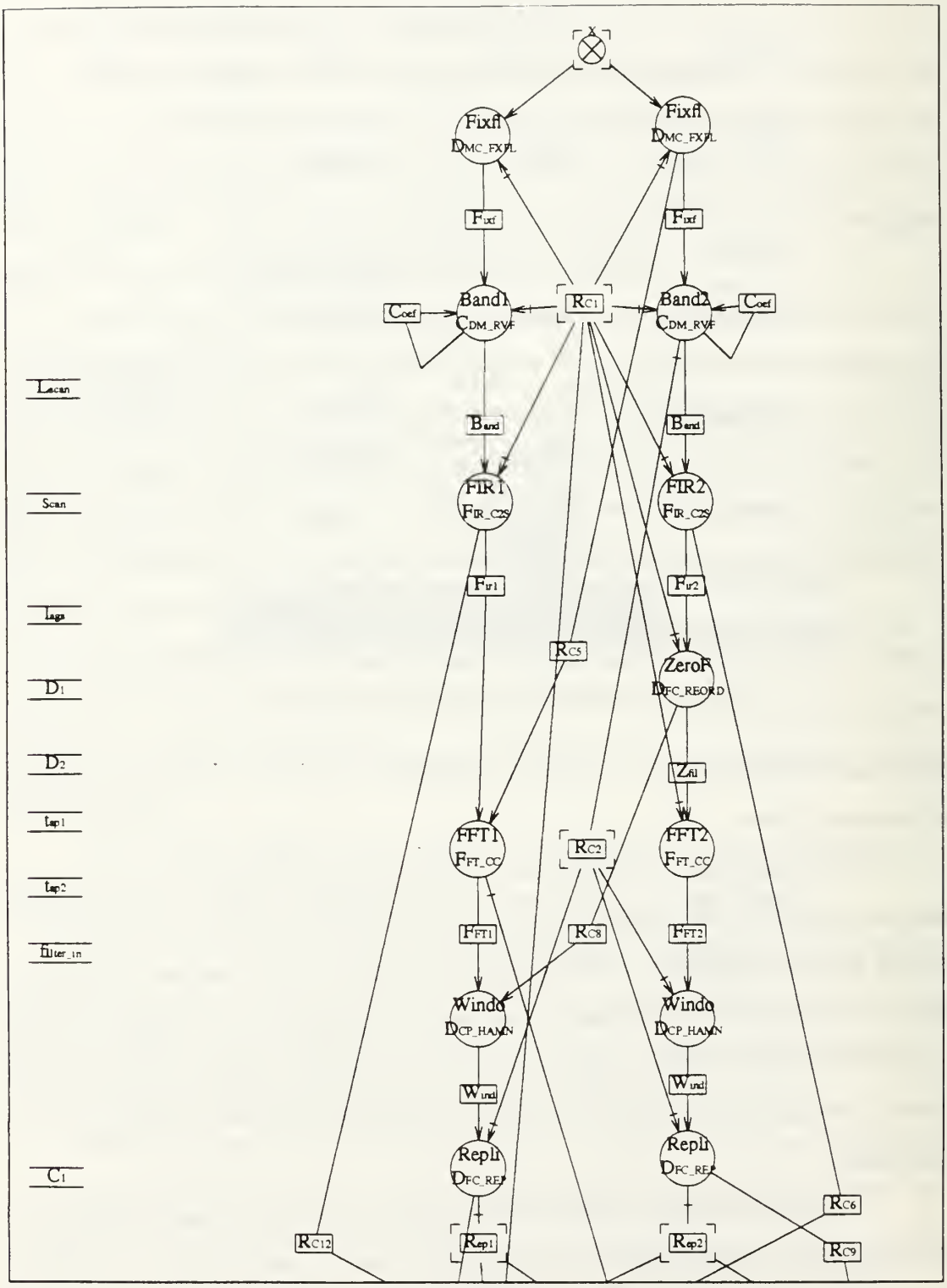


Figure 4.5: Restructured Correlator Graph (Part 1).

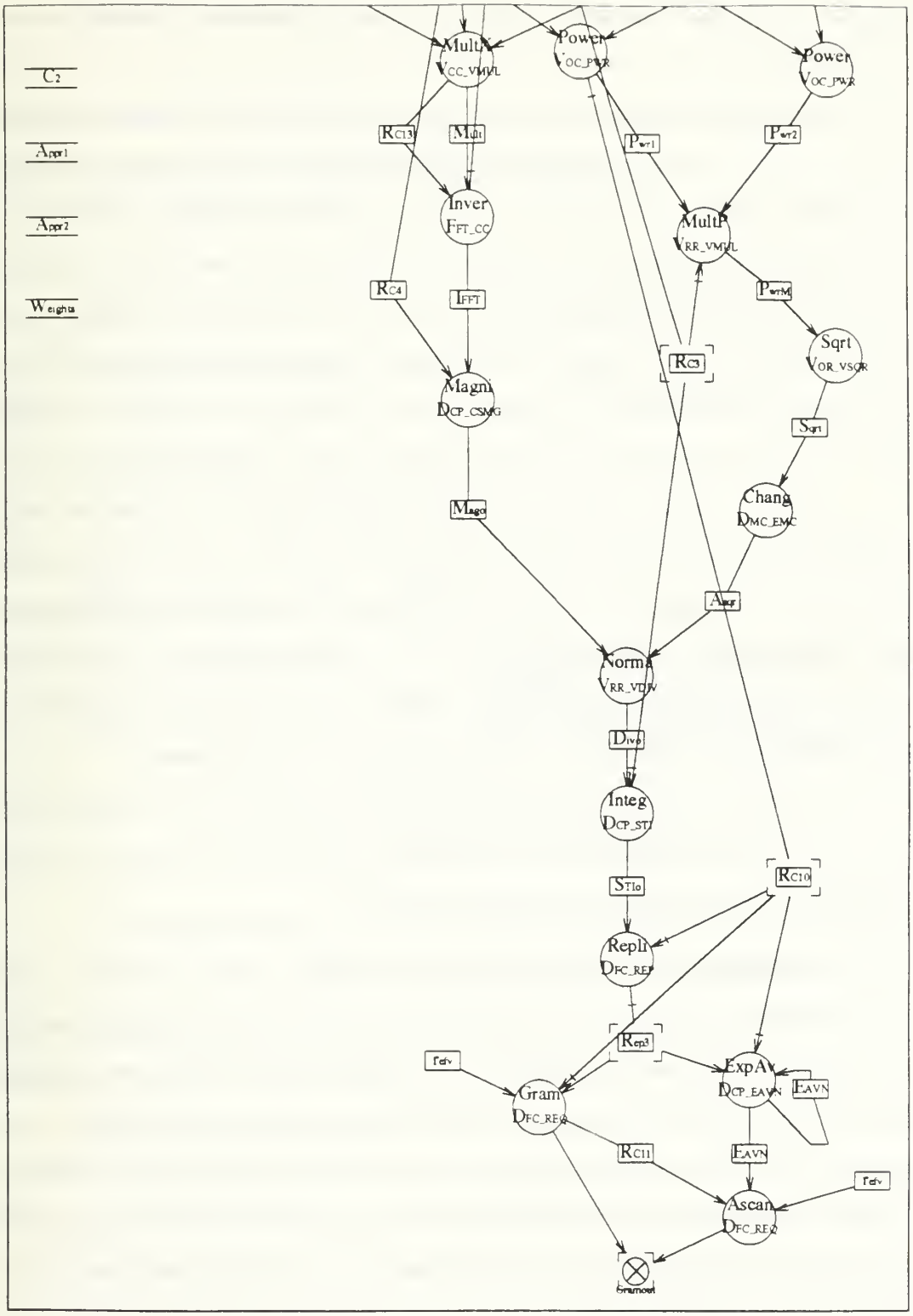


Figure 4.5: Restructured Correlator Graph (Part 2).

each node execution, but the number produced may be higher depending on node execution rates within the graph.

The local queue structure which acts as the trigger queue must also be initialized to mode *trigger*. If it is of kind family, as in queue *RC1* of the restructured correlator graph, the number of members must be stated. For *RC1*, this was <1..8>. Finally, the number of tokens to be placed on the queue prior to execution of the graph is stated in the *initial values* section. This section is seldom used in normal data-flow, since a prior knowledge of the data to be processed is rare. In trigger queues, however, dependent nodes may be allowed a finite number of executions prior to delays being introduced. The number of tokens placed in the initial values section will allow this finite number of executions to occur.

Threshold, read and *consume* amounts for the dependent node are specified at the input port. These tokens may be present initially, or as a result of the execution of a triggering node. Once threshold is achieved, the dependent node will begin consuming tokens at a rate specified by the *consume* amount. *Read* amount defaults to the consume amount if not specified, and so need not be altered. The dependency arc generation programs specify both *threshold* and *consume* amounts for all trigger queues.

The queue itself must be modified to reflect the mode of operation, *trigger* in this case, the number of family members if it is of type family, and the initial values to be placed on the queue(s), if any, at the start of program execution.

2. RC Restructuring of the Active Sonobuoy Graph

The correlator graph, with its flat, non-hierarchical structure using only single nodes, posed no difficulties in implementing the trigger queues. The active sonobuoy graph, however, possesses a hierarchical structure using subgraphs and chains, in both single and family types, in addition to extensive use of family types for queues, ports and nodes. While providing a very readable graph topology, and reduced SPGN code, these structures create formidable obstacles to the introduction of irregular structures such as trigger queues.

Figure 4.6 below is the cylinder mapping of the Active Sonobuoy graph using the mapping and restructuring programs. Input files and selected outputs for the restructuring program are included as Appendix J. All times are approximate. Eight processors were used, and form the columns of the cylinder surface mapping.

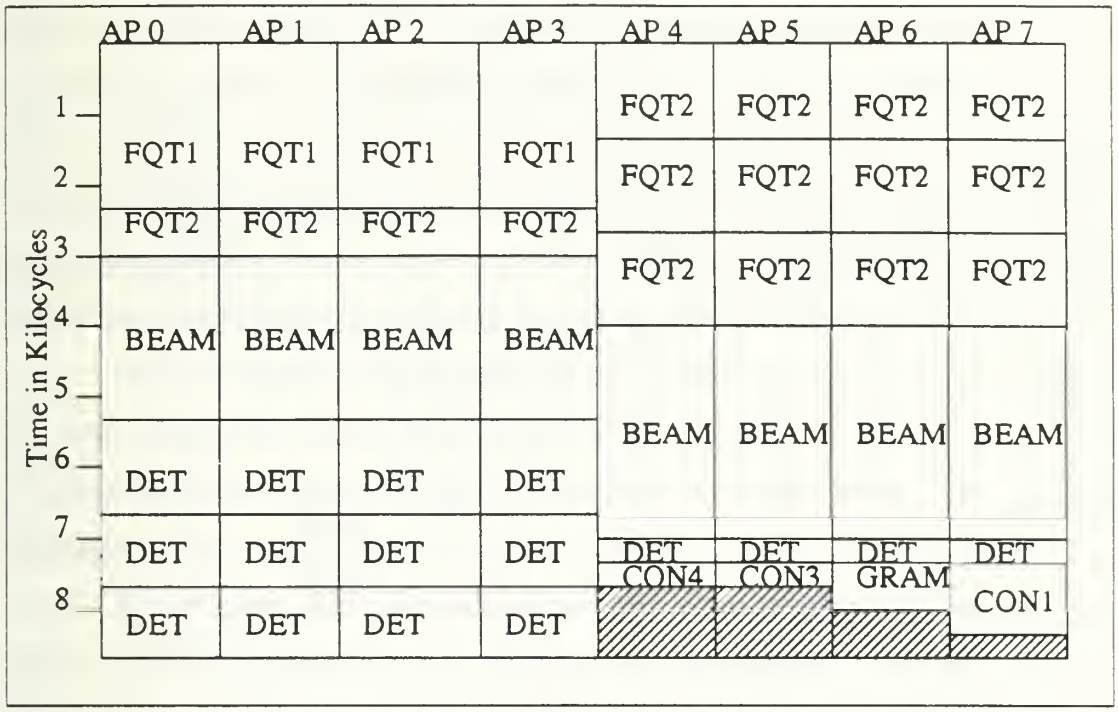


Figure 4.6: Active Graph Cylinder Mapping

This mapping resulted in a restructured graph with the topology seen in Figure 4.7 below. Comparison against Figure 3.7 indicates some of the actions necessary to implement the trigger queuing on this type of hierarchical graph structure. The complete graph topology is given in Appendix N, and the corresponding SPGN in Appendix O.

One problem encountered with the family subgraph, node and chain kinds used in the graph, is that either all members produce a triggering pulse, or none does. As an example, assume a 16 member family, such as Active Detect subgraph in the Active graph. If a particular member of the family is required to trigger another node, all members of the family must be modified with a PIP_OUT structure to produce a trigger pulse. This, in turn,

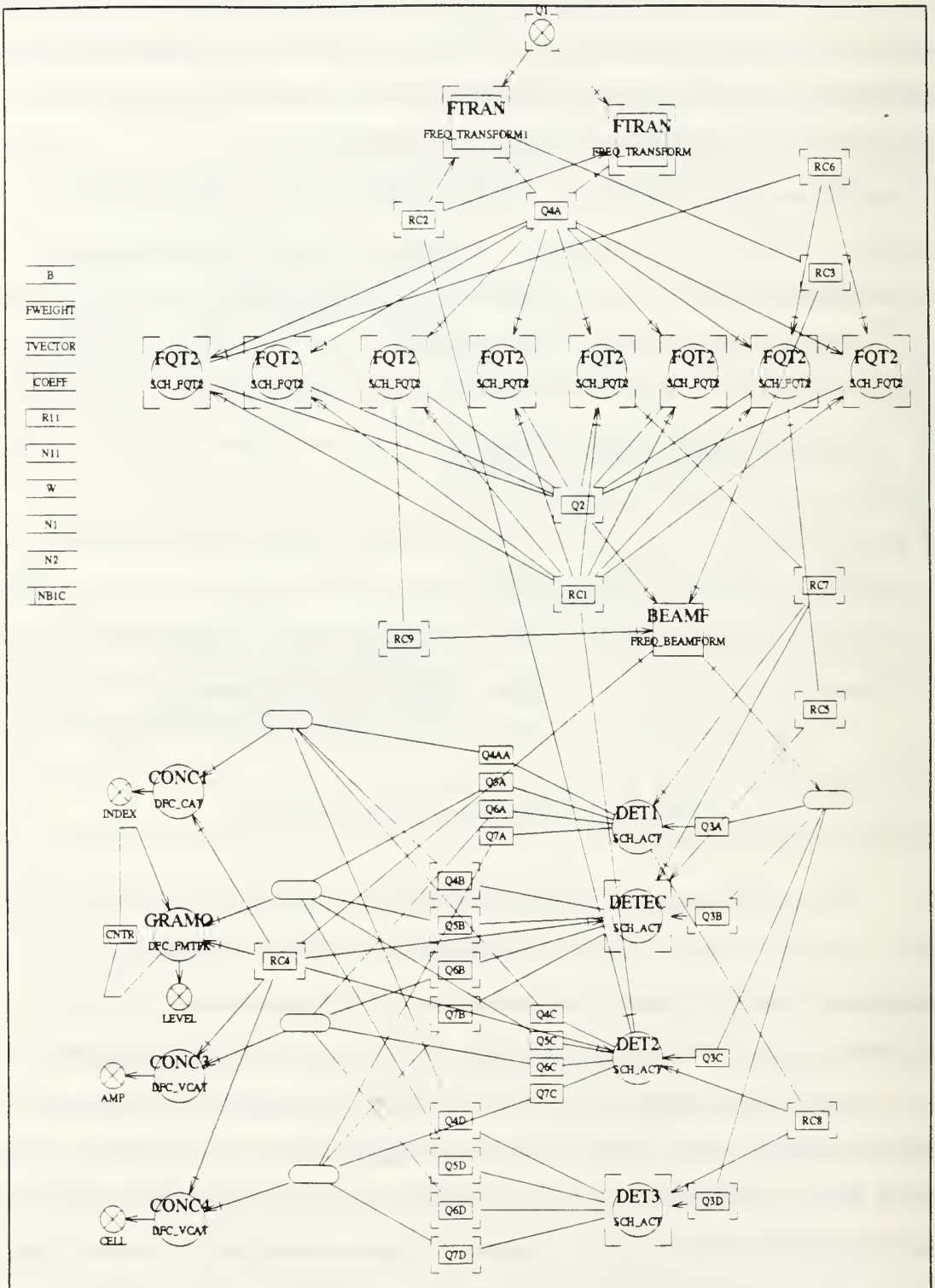


Figure 4.7: Restructured Active Sonobuoy Graph

necessitates addition of the local queues and ports to direct the trigger pulse to the appropriate dependent node. Failure to link each of the queues created, resulted in a simulation file generation error.

Many variations on indexing schemes were attempted in an effort to maintain the full family structure of each node and subgraph. This included generating arrays of pulses, an exhaustive combination of indexing systems, and use of input and output constructors. Constructors either create a family from disjoint objects, or distribute a family to disjoint objects [Ref. 7:p. 3.63-3.66]. Ultimately, the only viable alternative for implementation of trigger queues within a hierarchical graph structure is to separate those family members either generating or receiving triggering pulses from the other members of the family.

While not aesthetically pleasing, the procedural steps involved in separating the root level objects are simple. Separation involves the creation of additional objects, initialization of those objects, and finally, connection to provide data flow identical to the original structure.

The additional objects introduced must be created using the *introduce* procedure detailed in [Ref 9:p. 7-8], not by *replication* or *%COPY* commands. The objects created using these commands are linked to the original object so that changes, such as the introduction of PIPs, are reflected back to the original object. Once the new objects are created, they must be named using a distinct identifier, not that of the parent object.

Since subgraphs may be saved independently of the root graph, the original subgraph may be used as a template for the new subgraph objects. Once the new subgraph object is created, the original subgraph template may be read in and the new subgraph given a distinct name to differentiate it from the parent.

Initialization of nodes can be accomplished by first copying the NEPs from the parent, then adding the PIP_IN or PIP_OUT parameters. The $INx and $OUTx macros are recommended for the PIP procedures [Ref. 1]. The local index for the node is then set at 'I=x..y', where x and y are the first and last family members now represented by the new

object. Use of the 'I=' for each object allows the source and sink queues to remain referenced to a single local index, 'I'.

Subgraph indexing is performed as in separation of nodes, and the root level subgraph NEPs may be copied from the parent. The PIP_IN and PIP_OUT NEPs can only be introduced while editing a node. The subgraph incorporates the triggering pulses in the same manner as the actual data flow, via node to node pathways. If necessary, unchaining and separation of nodes within a cluster must take place in order to introduce the trigger queue.

The connections into, and out of, the new series of objects representing the family must be of type family. The NEP and mselector sections may be copied from the parent queue using the copy facility of GRED as outlined in [Ref 7:p. 3.32-3.34].

When a connection is started or terminated on the root level subgraph icon, an actual input or output queue is created at the subgraph level to provide the connection for the new data flow. Actual I/O queues must be connected to input and output queues, not to nodes themselves.

3. RC Restructuring of the Passive Sonobuoy Graph.

Restructuring of the Passive Sonobuoy graph was not attempted due to the problems encountered with the Active Sonobuoy graph, and the dissimilar execution rates of nodes in the graph.

V. SIMULATION AND PERFORMANCE

A. SIMULATION IN THE EWS FRAMEWORK

With the information provided by *gas*, the hardware configuration can be decided, and a dynamic simulation can begin. The hardware parameters are supplied to the simulator by the configuration file (*.cf by convention). This file can be written using a text editor, or by use of the Machine configuration EDitor (*med*). The required elements to be supplied by the configuration file are number of Arithmetic Processors (AP), number of Global Memory modules (GM) and number of Input/Output Processors (IOP). The other EMSP modules mentioned in Rice [Ref. 1:p. 2] are accounted for in the simulator and are superfluous here.

Med provides a graphical, building block approach and places realistic limits on the hardware configuration. For instance, *med* will not permit more than four functional elements on any backplane port and will not permit dissimilar FEs to be stacked. Backplane sizes available are four, eight and sixteen port. The *med* User's Guide [Ref. 17] fully explains this useful application.

Dynamic simulation is invoked using the *ets++* command followed by the appropriate suffixes and file names as described in [Ref. 7:p. 11.1-11.23]. The time the simulation should run in simulated machine seconds must also be specified. Other options for the command line include the trace-breakpoint-reset (*tbr*) language calls, use of simplified or detailed IOP models, specification of nodes and queues to be reported, and types of processors to simulate (SEM B or SEM E).

The *tbr* language calls allow certain hardware events to be traced and recorded. Snapshots of the system may also be requested at specified times, rather than by event. An example of an event recording is the execution of a node on a particular processor. In this case the processor may be specified, but not the node since execution of all nodes appear identical to the hardware. The node being executed is reported however, so creative use of *grep* and *awk* UNIX commands can usually provide the required data.

For example, the performance parameters reported in this chapter required knowledge of exact times at which the output queues were written. Use of tbr will currently only provide the times data arrives on the DTN for a specific queue, regardless of its origin [Ref. 7:p. B-tbr]. As seen in Figure 5.1, the origins are annotated, but must be extracted from the

```
1656444 GM(GM:0).DTNIN(WQ) 18
1657162 GM(GM:3).DTNIN(WQ) 19
1675408 GM(GM:1).DTNIN(WQ) 20
1681457 GM(GM:0).DTNIN(WQ) 0
1684816 GM(GM:2).DTNIN(WQ) 1
1695323 GM(GM:1).DTNIN(WQ) 20
1712905 GM(GM:1).DTNIN(WQ) 20
1730488 GM(GM:1).DTNIN(WQ) 20
1749512 GM(GM:2).DTNIN(WQ) 21
1758378 GM(GM:3).DTNIN(WQ) 34
1759832 GM(GM:4).DTNIN(WQ) 35
1775238 GM(GM:0).DTNIN(WQ) 22
1776692 GM(GM:4).DTNIN(WQ) 28
1779217 GM(GM:5).DTNIN(WQ) 2
1796077 GM(GM:1).DTNIN(WQ) 3
1961457 GM(GM:0).DTNIN(WQ) 0
1964816 GM(GM:2).DTNIN(WQ) 1
2241457 GM(GM:0).DTNIN(WQ) 0
```

Figure 5.1: Portion of Sample *ets++* Output using *tbr*

entire output file. The first UNIX command, *grep 'DTNIN(WQ) 2' | grep -v ' 2[0-9]' > outfile*', yields the desired lines from the file. This command extracts only the writes to queue 2, and excludes those to queues 20 through 29. This can be seen in Figure 5.2 below.

To extract only the times, '*awk '{print $1}' outfile > timesfile*' may be used. This will provide an ASCII file which may be used as input to a graphing program such as MATLAB.

Detailed IOP models of both SEM B and SEM E systems are provided and can yield additional insight to problems stemming from queue overloads and bus contention. In addition to the simple IOP parameters, FIFO statistics are given, data words lost are

```
1779217 GM(GM:5).DTNIN(WQ) 2
2899217 GM(GM:5).DTNIN(WQ) 2
4019217 GM(GM:5).DTNIN(WQ) 2
5139217 GM(GM:5).DTNIN(WQ) 2
6259217 GM(GM:5).DTNIN(WQ) 2
7379217 GM(GM:5).DTNIN(WQ) 2
8499217 GM(GM:5).DTNIN(WQ) 2
9619217 GM(GM:5).DTNIN(WQ) 2
10739217 GM(GM:5).DTNIN(WQ) 2
11859217 GM(GM:5).DTNIN(WQ) 2
12979217 GM(GM:5).DTNIN(WQ) 2
14099217 GM(GM:5).DTNIN(WQ) 2
15219217 GM(GM:5).DTNIN(WQ) 2
16339217 GM(GM:5).DTNIN(WQ) 2
```

Figure 5.2: Sample Grep Output File

tabulated, and average delay of incoming data, setup termination and execution completion are noted [Ref. 7:p. 11.4-11.6]. The default for the simulator is a SEM B (7 MHz), simplified IOP model.

An additional tool for dynamic analysis is the Simulator Performance Interface (*spi*), which links to a running *ets++* simulation via a software socket, and provides a graphical display of various simulation outputs. The *spi* application is well documented in [Ref. 17] and is a useful tool for simulation analysis. Hardcopy output of simulation runs is still required for reporting of all system language (SL) messages, DTN and CBUS activity, specific node execution parameters such as setup, breakdown and pending times, and other aspects of performance analysis.

B. PERFORMANCE MEASUREMENTS.

The ability to produce output as rapidly as input arrives is a simplistic measure of throughput. Normalizing the output rate to the input rate, yields normalized throughput, which will be unity so long as the system is able to process the data without delay. For graphs restructured by the RC approach this may decrease slightly since nodes which

would be scheduled for execution by FCFS are delayed in accordance with the structure imposed by the cylinder mapping. Fragmented cylinder mapping sacrifices the unused area of the cylinder for the ordering imposed by it.

Another method by which to measure throughput is comparison of the period between output queue writes to the period between input queue writes. Normalization of output period to input period is the normalized output period, which, again, will be unity so long as the system is not overloaded. Here again, RC restructured graphs may experience a slightly higher normalized output period since some nodes are being delayed, even when data has reached threshold.

Load is the percentage of maximum achievable throughput the system would have to maintain for the input data rate being supplied. Since the inclusion of trigger queues in data-flow graphs does not increase graph execution time, the maximum achievable throughput for restructured graphs is identical to the parent graphs.

Coefficient of variation (COV) defines the standard deviation of a data set as a function of the mean of the data set. In this way, two data sets of different magnitudes can be compared for relative consistency from data point to data point within the set. A lower coefficient of variation indicates less variation in output from one sample to the next. For the signal processing graphs, a lower COV indicates that the period from output write to output write is more regular than it would be with a higher COV. RC restructuring should significantly lower the COV for an application since run-time node ordering is now more structured.

C. PERFORMANCE COMPARISONS

The two areas of performance which the Revolving Cylinder algorithm attempts to improve are throughput and execution consistency as measured by latency. Throughput is measured as the number of output queue writes multiplied by the size of the data packet written divided by the time interval. Execution consistency is measured using a coefficient of variation for each data rate.

1. Correlator Graph

The baseline Correlator graph shown in Figure 2.6 and restructured Correlator graph shown in Figure 4.6 were simulated at varying input data rates from 40960 total words per minute (wps) to 1638400 total wps. Using the maximum throughput of 1,381,783 wps calculated in Chapter 3, this represents a range of 3% to 118% loading.

a. Output Period

Normalized mean times between writes are given for each output queue in Figures 5.3 and 5.4 below. The slightly longer times observed for the restructured graph are attributable to the fragmented nature of the revolving cylinder. Unused processor time is represented by the hashed portions of the cylinder diagram (Figure 4.4). Both implementations deviate from the ideal of 1.0 at approximately the same loading (45%).

b. Throughput and Utilization

Reinforcement of the results of the previous paragraph is found by comparing the rate at which data is written to the output queue as a function of loading. Figure 5.3 verifies the expected decrease in throughput expected of the restructured graph.

c. Coefficient of Variation

The correlator graph normalized coefficients of variation are shown in Figures 5.4 and 5.5. The normalized coefficients of variation indicate that the restructured graph clearly provides a more uniform rate of output queue writes than the baseline structure. Normalized standard deviation is also provided in Figures 5.6 and 5.7.

2. Active Sonobuoy Graph

The Revolving Cylinder algorithm as currently implemented, may introduce a deadlock condition into data-flow graphs operating under the ECOS methodology. To mediate potential bottlenecks within the data flow of ECOS graphs, queues which exceed capacity initiate a inhibition mechanism for nodes preceding them in the data flow. Delays introduced by RC may trigger these mechanisms and therefore indirectly inhibit execution

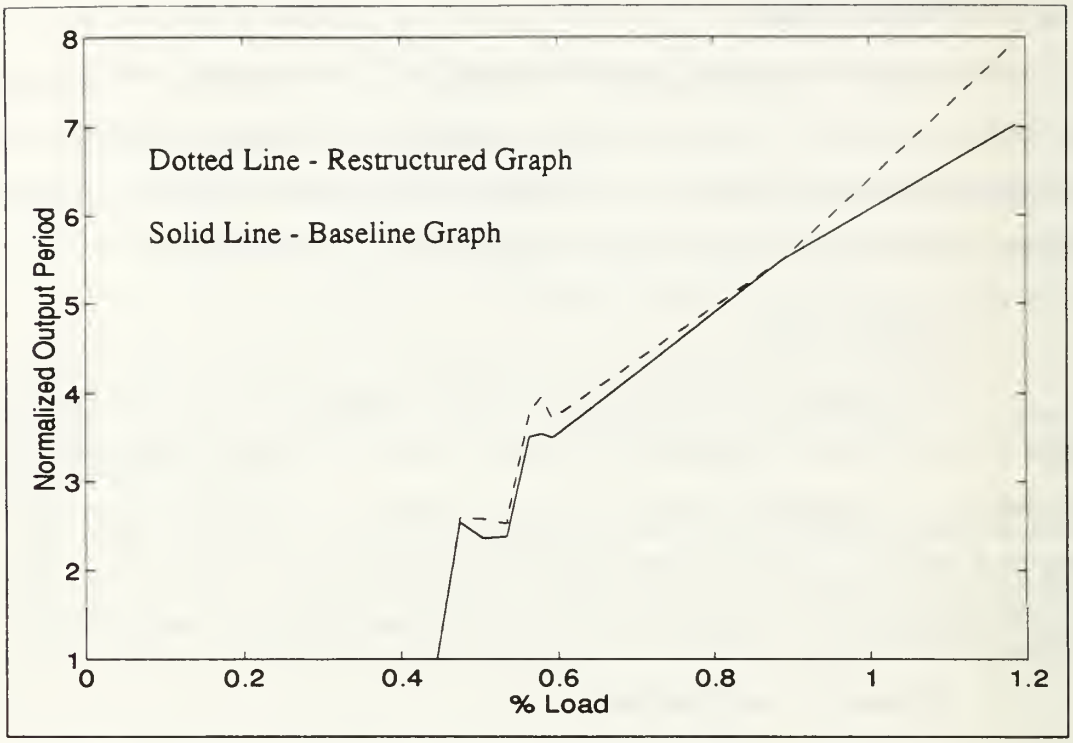


Figure 5.3: Normalized Output Period for Gramout

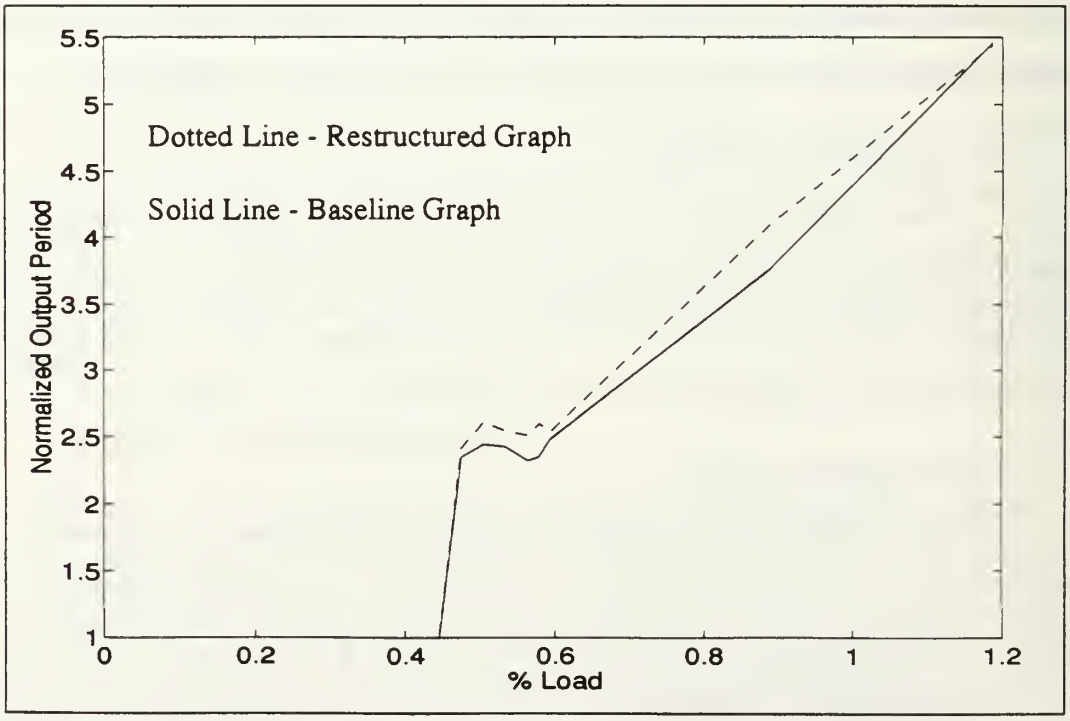


Figure 5.4: Normalized Output Period for Ascan

of nodes whose triggering is required to alleviate the data backlog. This, in effect, stops execution of the graph completely since a circular dependency is created.

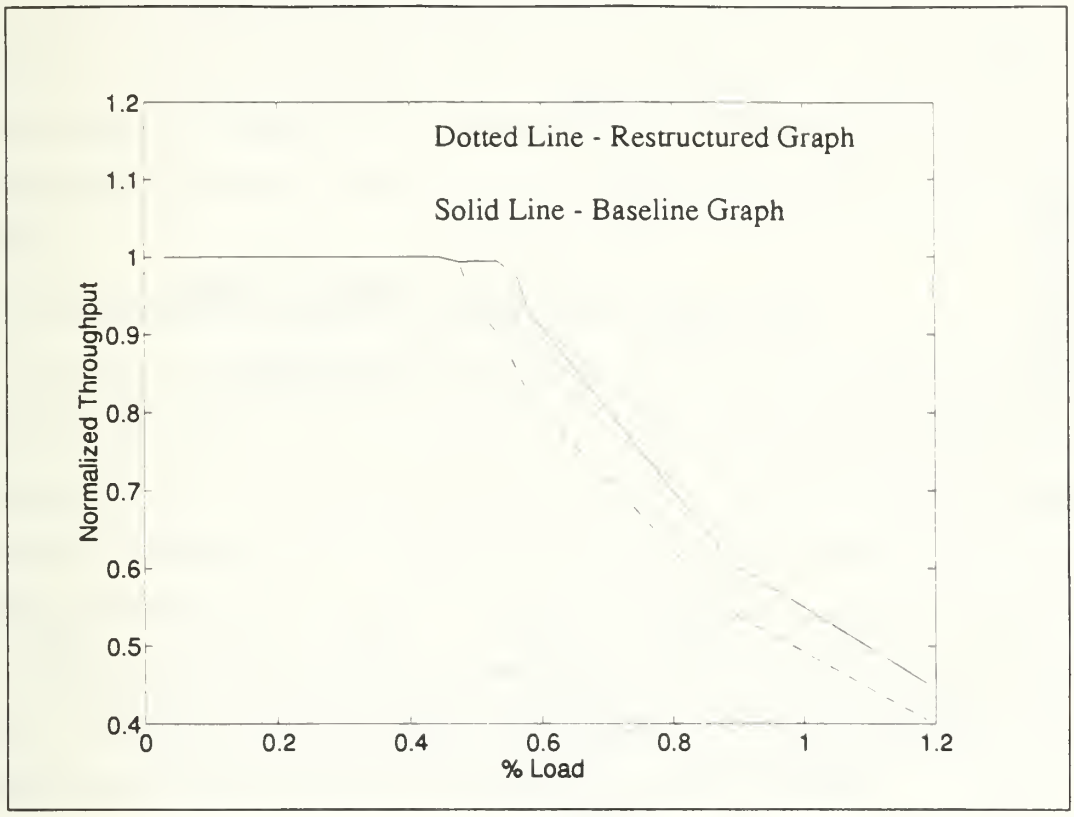


Figure 5.5: Correlator Graph Throughput Versus Loading

The production of Stop Node Data Transmissions (SNDT), which are a stopgap measure implemented by the ECOS system when input queues reach capacity (32 kilobytes or 8 times threshold, whichever is greater), indicates the data is accumulating on the various local queues more rapidly than it is being produced at the output nodes. The Restructured Active Sonobuoy graph experienced SNDT production very early in the simulation, and due to the circular dependencies introduced, was unable to continue processing.

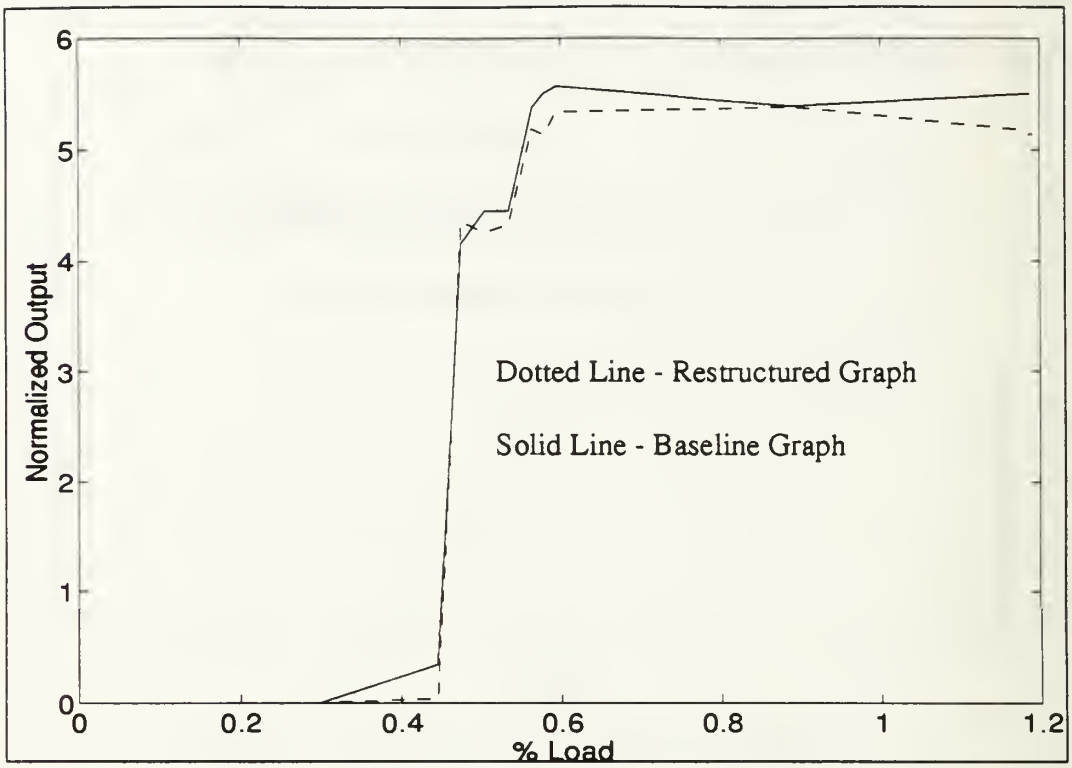


Figure 5.6: Correlator Graph Coefficients of Variation for Gramout

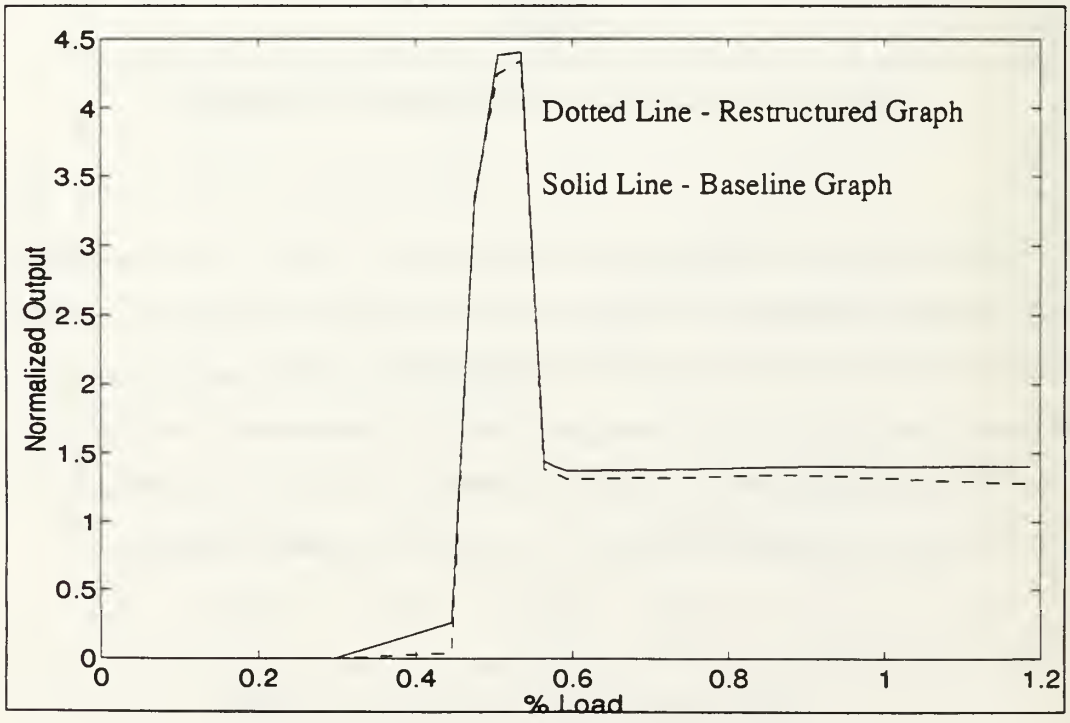


Figure 5.7: Correlator Graph Coefficients of Variation for Ascan.

VI. CONCLUDING REMARKS

Use of the ECOS Workstation System (EWS) to introduce the dependency arcs required in implementing the Revolving Cylinder algorithm can be accomplished with varying degrees of difficulty. Once implemented, the advantages of RC restructuring in reducing the inconsistancy inherent in First Come, First Serve (FCFS) scheduling are evident.

A. PRACTICALITY OF RC SCHEDULING UNDER EWS

The most significant obstacle to the use of trigger queues in EWS is the family structure of some affected nodes and subgraphs. The requirement to separate affected nodes or subgraphs at the root level of the graph heirarchy in order to permit the irregular placement of trigger queues is tedious and can create a 'busy' graphical picture when implemented using *gred*.

Perhaps equally important is the current failure to account for Queue Over Capacity (QOC) correction in ECOS and EMSP. This mechanism must be accounted for in order to generate dependencies which do create a circular dependency condition.

Once implemented, EWS provides an excellent capability for analyzing and evaluating the potential improvements sought using RC. The *spi* interface can provide graphical performance results, and the remaining parameters are available as normal *gas* and *ets++* output.

B. RECOMMENDATIONS FOR FUTURE RESEARCH

To prevent the deadlock seen while implementing RC on graphs with differing node execution rates, such as the Active Sonobuoy graph, another mechanism must be incorporated into the mapping and restructuring programs. This mechanism must account for the known compile-time allocation of memory for queue buffers, and check to ensure that the dependencies being generated do not force any local queues over capacity. This

may be approached by comparison of the magnitude of the indexing to the multiplicity of the nodes in the graph being restructured.

In some cases, node execution rates may be equalized by modification of *produce* and *consume* amounts for data-flow between adjacent nodes with different execution rates, or, by minor graph modifications. An example of the latter is shown in Figure 6.1, where the beamforming area of the Active Sonobuoy graph has been demultiplexed and passed to 8 *BEAMFORMER* subgraphs. The output of these 8 subgraphs is then multiplexed and passed on to the ACTIVE DETECT subgraphs as before. Using this method, the BEAMFORMER subgraph now executes at the same rate as the remaining portions of the graph, rather than multiple sequential executions. The modified portion of the SPGN is included as Figure 6.2.

Current RC implementation was attempted using only dependency arcs. Parallel or serial chaining should be explored as an alternative and complementary method of enforcing compile-time scheduling.

The deviations in throughput found in comparison of the results reported in this thesis to those of [Ref. 13], may be attributable to simulator dissimilarities. The setup, breakdown and execution times utilized by EWS are precisely defined by the PID specifications, and by the data flow within the graph topology. The input quantities specified in [Ref. 13] were approximations based on available information.

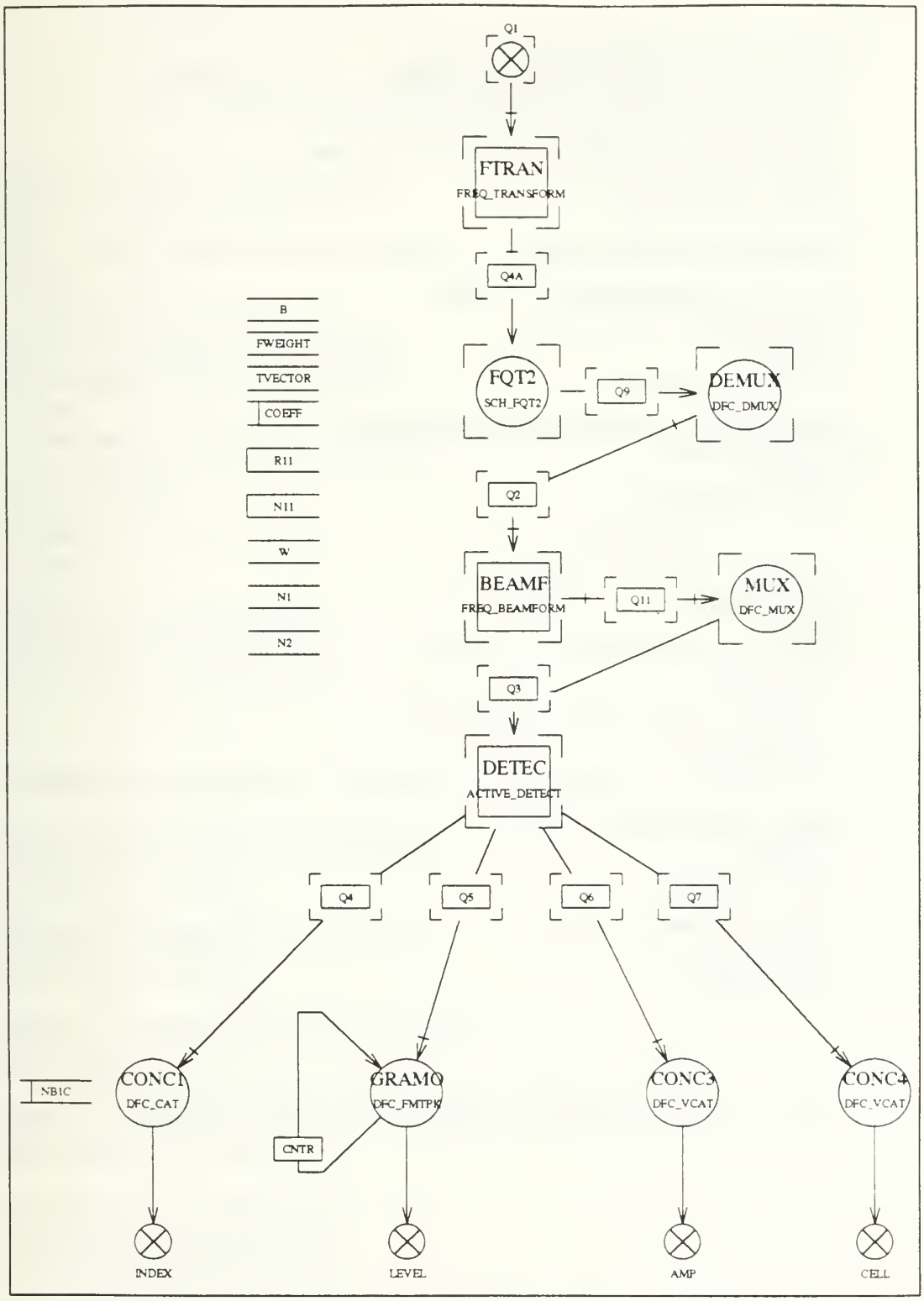


Figure 6.1: Modified Active Sonobuoy Graph

```
%QUEUE([1..NC,1..8]Q2: CFLOAT)%% FAMILY OF NC SPECTRA
%QUEUE([1..NB]Q3: FLOAT)%% FAMILY OF BEAM MAGNITUDES
%QUEUE([1..NB]Q4: INT)%% FAMILY OF NB PEAK INDICES
%QUEUE([1..NB]Q7: INT V_ARRAY(NF))%% FAMILY OF NB BIN NUMBERS
%QUEUE([1..NB]Q5: INT)%% FAMILY OF NB PEAK LEVELS
%QUEUE([1..NB]Q6: FLOAT V_ARRAY(NF))%% FAMILY OF NB AMPLITUDES
%QUEUE([1..16]Q4A: CFLOAT)%% IIR FILTER OUT
%QUEUE([1..NC]Q9: CFLOAT)
%QUEUE([1..8,1..16]Q11: FLOAT)

%% TOPOLOGY section (%NODE, %SUBGRAPH)

%SUBGRAPH([I=1..NB]DETECTGRAPH = ACTIVE_DETECT
GIP = NF,NAVG,NB1,NLEVEL,TVECTOR,
C,R,WIDTH,TH,IA
VAR = AVLV
INPUTQ = [I]Q3
OUTPUTQ = [I]Q4,[I]Q5,[I]Q6,[I]Q7)

%SUBGRAPH([I=1..NC/4]FTRANSFORMGRAPH = FREQ_TRANSFORM
GIP = N,TS,RATE,NFFT,NF,
NW,FWEIGHT
VAR = FC,[1..4]COEFF,RFFT,B
INPUTQ = [(I-1)*4+1..(I-1)*4+4]Q1
OUTPUTQ = [(I-1)*4+1..(I-1)*4+4]Q4A)

%NODE([I=1..NC]FQT2PRIMITIVE = SCH_FQT2
PIP_IN = R11,N11
PRIM_IN = N1,N1,N2,0,1,
B,NW,N2,W,
[I]Q4ATHRESHOLD = 1920VARIABLE CONSUME = 8*(N11/R11)
PRIM_OUT = [I]Q9)

%SUBGRAPH([J=1..8]BEAMFORMERGRAPH = FREQ_BEAMFORM
GIP = NF,NC,NB
VAR = [1..NB]BWEIGHT
INPUTQ = [1..NC,J]Q2
OUTPUTQ = [J,1..16]Q11)

%NODE([I=1..16]DEMUXPRIMITIVE = DFC_DMUX
PRIM_IN = NF,8,
[I]Q9THRESHOLD = 8*NF
PRIM_OUT = [I,1..8]Q2)

%NODE([I=1..16]MUXPRIMITIVE = DFC_MUX
PRIM_IN = NF,8,
[1..8,I]Q11THRESHOLD = NF
PRIM_OUT = [I]Q3)
```

Figure 6.2: Selected SPGN for Modified Active Sonobuoy Graph

APPENDIX A: CORRELATOR GRAPH SPGN†

```
%%
%% c-base.g: %Z% %P% %I% %G%
%%
%%*************************************************************%%
%% %%
%% GRAPH 'E006 ' %%
%% SPGN generated from GRED %%
%% Mon May 17 22:40:08 1993 %%
%% %%
%%*************************************************************%%
%GRAPH(E006
 GIP = SR: dfloat,
 F: dfloat,
 TC: dfloat,
 STI: int
 INPUTQ = [1..2]X: FIXED(2)
 OUTPUTQ = [1..2]Gramout: INT
)


%% DECLARATIONS section (%GIP, %VAR, %QUEUE)

%GIP(Lscan: INT%% 4 Long scan length
 INITIALIZE TO 16384)

%GIP(scan: INT%% 4 Short scan length
 INITIALIZE TO 4096)

%GIP(lags: INT%% number of correlation lags
 INITIALIZE TO 513)

%GIP(C1: DFLOAT ARRAY(3)
 INITIALIZE TO {3 OF 0.0E0})

%GIP(C2: DFLOAT ARRAY(8)
 INITIALIZE TO {8 OF 0.0E0})
```

† Created from original file received from TRW.

```
%GIP(Weights: FLOAT ARRAY(26)
 INITIALIZE TO {26 OF 1.0E0})

%GIP(D1: INT%% Decimations for filter
 INITIALIZE TO 2)

%GIP(D2: INT
 INITIALIZE TO 2)

%GIP(tap1: INT
 INITIALIZE TO 7)

%GIP(tap2: INT%% # of taps for filter
 INITIALIZE TO 19)

%GIP(filter_in: INT%% input to filter
 INITIALIZE TO D1*(D2*(scan-1)+tap2-1)+tap1)

%GIP(Appr1: FLOAT
 INITIALIZE TO 0.961E0)

%GIP(Appr2: FLOAT
 INITIALIZE TO 0.398E0)

%QUEUE(Fixflout1: DFLOAT)
%QUEUE(Band1out: DCFLOAT
 INITIALIZE TO 39 OF <0.0E0,0.0E0>)

%QUEUE(Band2out: DCFLOAT
 INITIALIZE TO 39 OF <0.0E0,0.0E0>)

%QUEUE(Fixflout2: DFLOAT)
%QUEUE(Fir2out: DCFLOAT)
%QUEUE(Fir1out: DCFLOAT)
%QUEUE(Zfilout: DCFLOAT)
%QUEUE(FFT2out: DCFLOAT)
%QUEUE(FFT1out: DCFLOAT)
%QUEUE(Wind1out: DCFLOAT)
%QUEUE(Wind2out: DCFLOAT)
%QUEUE(IFFTout: DCFLOAT)
%QUEUE(PwrMultout: DFLOAT)
%QUEUE(Sqrtout: DFLOAT)
```

```
%QUEUE(Asqrtout: DFLOAT ARRAY(1))
%QUEUE(Magout: DFLOAT)
%QUEUE(Divout: DFLOAT)
%QUEUE(STIout: DFLOAT)
%QUEUE(EAVNout: DFLOAT)
%QUEUE(Multout: DCFLOAT)
%QUEUE(Pwr1out: DFLOAT)
%QUEUE(Pwr2out: DFLOAT)
%QUEUE(Coeffptr1: INT
 INITIALIZE TO 1)

%QUEUE(Coeffptr2: INT
 INITIALIZE TO 1)

%QUEUE(EAVNfeed: DFLOAT
 INITIALIZE TO lags OF 0.0E0)

%QUEUE(refvect2: DFLOAT
 INITIALIZE TO lags OF 1.0E0)

%QUEUE([1..2]Rep2out: DCFLOAT)
%QUEUE([1..2]Rep1out: DCFLOAT)
%QUEUE([1..2]Rep3out: DFLOAT)
%QUEUE(refvect: DFLOAT
 INITIALIZE TO lags OF 1.0E0)


%% TOPOLOGY section (%NODE, %SUBGRAPH)

%NODE(Fixfl1PRIMITIVE = DMC_FXFL
 PRIM_IN = Lscan,
 [1]XTHRESHOLD = LscanREAD = LscanCONSUME = Lscan
 PRIM_OUT = Fixflout1)

%NODE(Fixfl2PRIMITIVE = DMC_FXFL
 PRIM_IN = Lscan,
 [2]XTHRESHOLD = LscanREAD = LscanCONSUME = Lscan
 PRIM_OUT = Fixflout2)

%NODE(Band1PRIMITIVE = CDM_RVF
 PRIM_IN = Lscan,Unused,1024,F,SR,
 Coeffptr1THRESHOLD = 1,
 Fixflout1THRESHOLD = LscanREAD = LscanCONSUME = Lscan
```

```
 PRIM_OUT = Band1out,Coeffptr1)

%NODE(FIR1PRIMITIVE = FIR_C2S
 PRIM_IN = filter_in,tap1,tap2,D1,D2,
 Weights,
 Band1outTHRESHOLD = filter_inREAD = filter_inCONSUME = Lscan
 PRIM_OUT = Fir1out)

%NODE(Band2PRIMITIVE = CDM_RVF
 PRIM_IN = Lscan,Unused,1024,F,SR,
 Coeffptr2THRESHOLD = 1,
 Fixflout2THRESHOLD = LscanREAD = LscanCONSUME = Lscan
 PRIM_OUT = Band2out,Coeffptr2)

%NODE(FIR2PRIMITIVE = FIR_C2S
 PRIM_IN = filter_in,tap1,tap2,D1,D2,
 Weights,
 Band2outTHRESHOLD = filter_inREAD = filter_inCONSUME = Lscan
 PRIM_OUT = Fir2out)

%NODE(ZeroFillPRIMITIVE = DFC_REORD
 PRIM_IN = scan,scan,3,2,256,
 767,
 Fir2outTHRESHOLD = scanREAD = scanCONSUME = scan
 PRIM_OUT = Zfilout)

%NODE(FFT2PRIMITIVE = FFT_CC
 PRIM_IN = scan/4,scan/4,0,1,
 ZfiloutTHRESHOLD = scanREAD = scanCONSUME = scan
 PRIM_OUT = FFT2out)

%NODE(FFT1PRIMITIVE = FFT_CC
 PRIM_IN = scan/4,scan/4,0,1,
 Fir1outTHRESHOLD = scanREAD = scanCONSUME = scan
 PRIM_OUT = FFT1out)

%NODE(Window1PRIMITIVE = DCP_HAMN
 PRIM_IN = scan/4,1,
 FFT1outTHRESHOLD = scanREAD = scanCONSUME = scan
 PRIM_OUT = Wind1out)

%NODE(Window2PRIMITIVE = DCP_HAMN
 PRIM_IN = scan/4,1,
```

```
FFT2outTHRESHOLD = scanREAD = scanCONSUME = scan
PRIM_OUT = Wind2out)

%NODE(Replicate2PRIMITIVE = DFC_REP
PRIM_IN = SCAN,2,
Wind2outTHRESHOLD = scanREAD = scanCONSUME = scan
PRIM_OUT = [1..2]Rep2out)

%NODE(Replicate1PRIMITIVE = DFC_REP
PRIM_IN = SCAN,2,
Wind1outTHRESHOLD = scanREAD = scanCONSUME = scan
PRIM_OUT = [1..2]Rep1out)

%NODE(MultXYPRIMITIVE = VCC_VMUL
PRIM_IN = scan,0,
[1]Rep1outTHRESHOLD = scanREAD = scanCONSUME = scan,
[1]Rep2outTHRESHOLD = scan
PRIM_OUT = Multout)

%NODE(PowerXPRIMITIVE = VOC_PWR
PRIM_IN = scan/4,
[2]Rep1outTHRESHOLD = scanREAD = scanCONSUME = scan
PRIM_OUT = Unused,Pwr1out)

%NODE(PowerYPRIMITIVE = VOC_PWR
PRIM_IN = scan/4,
[2]Rep2outTHRESHOLD = scanREAD = scanCONSUME = scan
PRIM_OUT = Unused,Pwr2out)

%NODE(InverseFFTPRIMITIVE = FFT_CC
PRIM_IN = scan/4,lags,1,257,
MultoutTHRESHOLD = scanREAD = scanCONSUME = scan
PRIM_OUT = IFFTout)

%NODE(MagnitudePRIMITIVE = DCP_CSMG
PRIM_IN = 4*lags,Appr1,Appr2,
IFFToutTHRESHOLD = 4*lagsREAD = 4*lagsCONSUME = 4*lags
PRIM_OUT = Magout)

%NODE(MultPowerPRIMITIVE = VRR_VMUL
PRIM_IN = 4,
Pwr2outTHRESHOLD = 4,
Pwr1outTHRESHOLD = 4
```

```
 PRIM_OUT = PwrMultout)

%NODE(SqrtPRIMITIVE = VOR_VSQR
 PRIM_IN = 1,
 PwrMultoutTHRESHOLD = 1
 PRIM_OUT = Sqrtout)

%NODE(ChangePRIMITIVE = DMC_EMC
 PRIM_IN = 1,
 SqrtoutTHRESHOLD = 1
 PRIM_OUT = Asqrtout)

%NODE(NormalizePRIMITIVE = VRR_VDIV
 PRIM_IN = lags,
 MagoutTHRESHOLD = lagsREAD = lagsCONSUME = lags,
 AsqrtoutTHRESHOLD = 1READ = 1CONSUME = 1
 PRIM_OUT = Divout)

%NODE(IntegratePRIMITIVE = DCP_STI
 PRIM_IN = lags,STI,STI,Unused,Unused,
 DivoutTHRESHOLD = lags*STI
 PRIM_OUT = Unused,Unused,STIout)

%NODE(Replicate3PRIMITIVE = DFC_REP
 PRIM_IN = lags,2,
 STIoutTHRESHOLD = lagsREAD = lagsCONSUME = lags
 PRIM_OUT = [1..2]Rep3out)

%NODE(GramPRIMITIVE = DFC_REQ
 PRIM_IN = lags,C2,
 [1]Rep3outTHRESHOLD = lagsREAD = lagsCONSUME = lags,
 refvectTHRESHOLD = lagsREAD = lagsCONSUME = 0
 PRIM_OUT = [1]Gramout)

%NODE(ExpAvgPRIMITIVE = DCP_EAVN
 PRIM_IN = 1,lags,TC,0,
 EAVNfeedTHRESHOLD = lagsREAD = lagsCONSUME = lags,
 [2]Rep3outTHRESHOLD = lagsREAD = lagsCONSUME = lags
 PRIM_OUT = EAVNout,EAVNfeed)

%NODE(AscanPRIMITIVE = DFC_REQ
 PRIM_IN = lags,C2,
 EAVNoutTHRESHOLD = lags,
```

```
refvect2THRESHOLD = lagsREAD = lagsCONSUME = 0
PRIM_OUT = [2]Gramout)

%ENDGRAPH
```

APPENDIX B: CORRELATOR GRAPH COMMAND PROGRAM†

```
%INITCOMPROG()

int STI = 4;
floatF = 500.0;
floatSR = 1000.0;
floatTC = 50.0;

IO_PROC_IDiop1,iop2,iop3,iop4;
GRAPH_IDG;
QUEUE_IDib[2],ob[2];

ib[0]=%CREATEQ(FIXED);
ib[1]=%CREATEQ(FIXED);
ob[0]=%CREATEQ(INT);
ob[1]=%CREATEQ(INT);
G = %START(E006
GIP = SR,F,TC,STI
INPUTQ = FAMILY(ib[0],ib[1])
OUTPUTQ = FAMILY(ob[0],ob[1])
PRIORITY = 2);

if (%SCODE) exit(1);

iop1 = %INITIO(IN1 INPUTQ = ib[0]);
iop2 = %INITIO(IN2 INPUTQ = ib[1]);
iop3 = %INITIO(OUT1 OUTPUTQ = ob[0]);
iop4 = %INITIO(OUT2 OUTPUTQ = ob[1]);
%STARTIO(iop1);
%STARTIO(iop2);
%STARTIO(iop3);
%STARTIO(iop4);

%PRINT(%TERM,1,G);

%ENDPROGRAM
```

† Created from original file received from AT&T Bell Laboratories, Whippany, N.J.

APPENDIX C: CORRELATOR GRAPH INPUT/OUTPUT FILE†

```
# @(#) /b/cm/emsp/sef/sccs/b2/eo/s.eo.ioproc.cf 1.2 3/5/91

# The format of the i/o procedure file is tabular; the information
# content is simple, so this should cause no problems, and it reduces
# the amount of baggage in the simulator for interpreting this file.

# NAME is the symbolic identifier of the i/o procedure named in
# the %INITIO statement in the command program; it must match exactly;
# case is significant.
#
# TYPE is one of INPUT, BIDIRECTIONAL, or OUTPUT
#
# DATA RATES for graph inputs are expressed in words/second
#
# THRESH, READ, CONSUME, and PRODUCE amounts are in words
#
# comments look like this, everything on a line to the right
# of the sharp sign is discarded

#ports
# nametypeoutput
#---------------
IN1INPUT1

# rateproduce
#------------
20482048

IN2INPUT1

# rateproduce
#------------
20482048

#ports
```

† Created from original file received from AT&T Bell Laboratories, Whippany, N.J.

nametypeinput
#----------
OUT1OUTPUT1

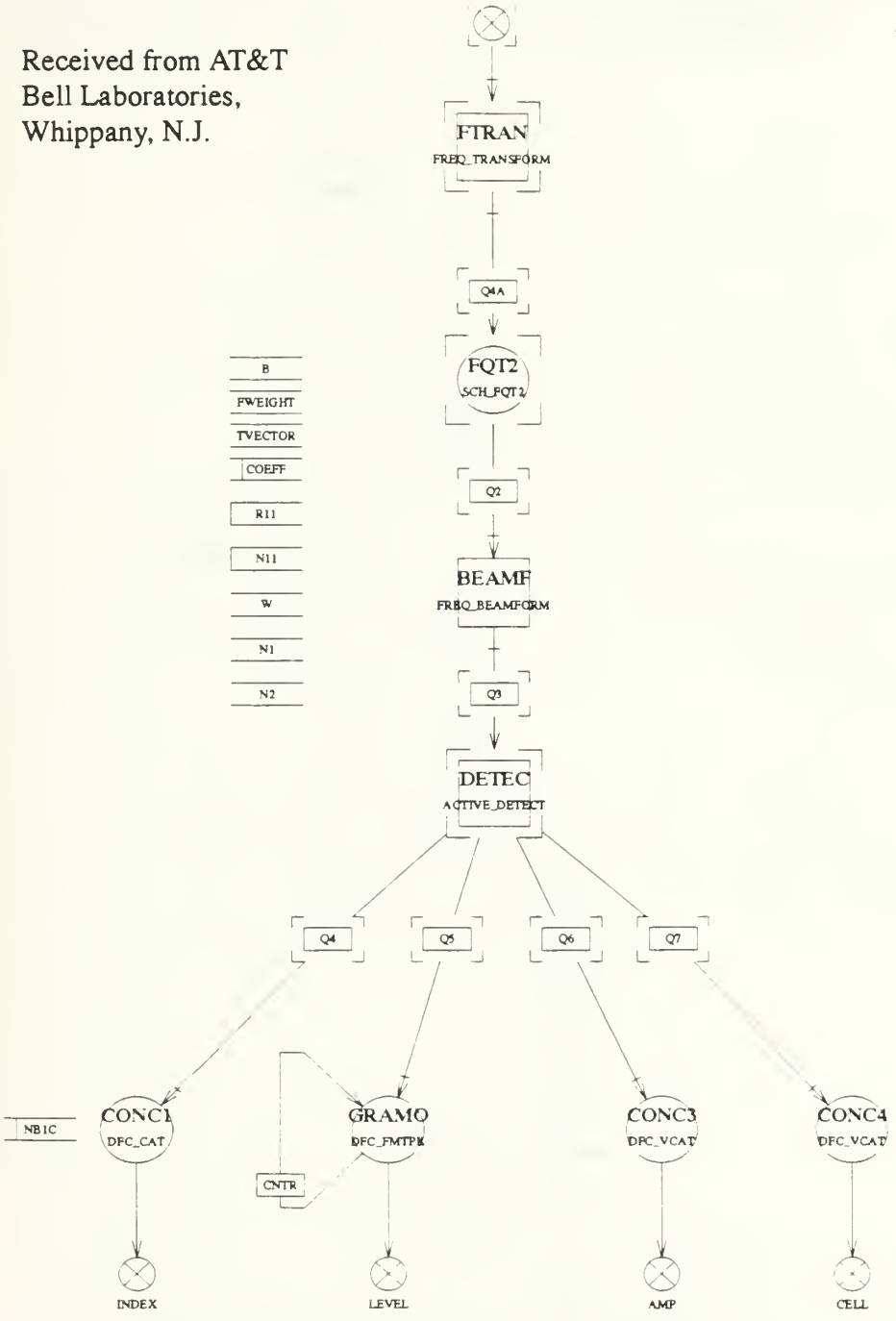
#threshreadconsume
#----------
513513513

OUT2OUTPUT1

#threshreadconsume
#----------
513513513

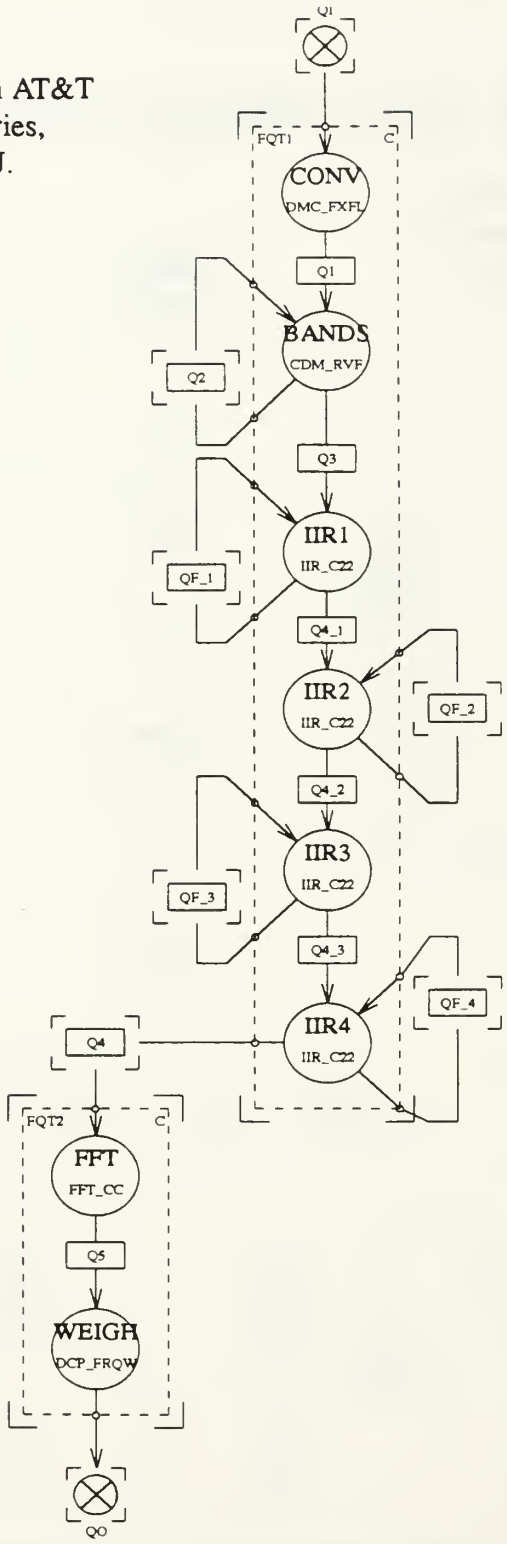
APPENDIX D: ACTIVE SONOBUOY GRAPH TOPOLOGIES

Received from AT&T
Bell Laboratories,
Whippany, N.J.



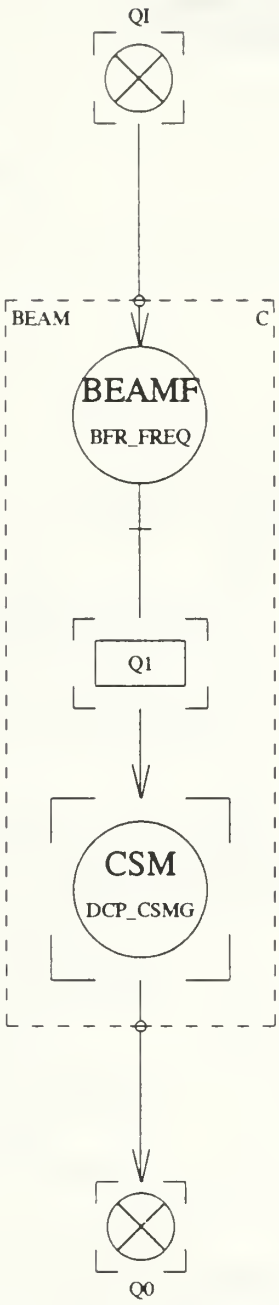
Active Sonobuoy Graph - Root Level

Received from AT&T
Bell Laboratories,
Whippany, N.J.



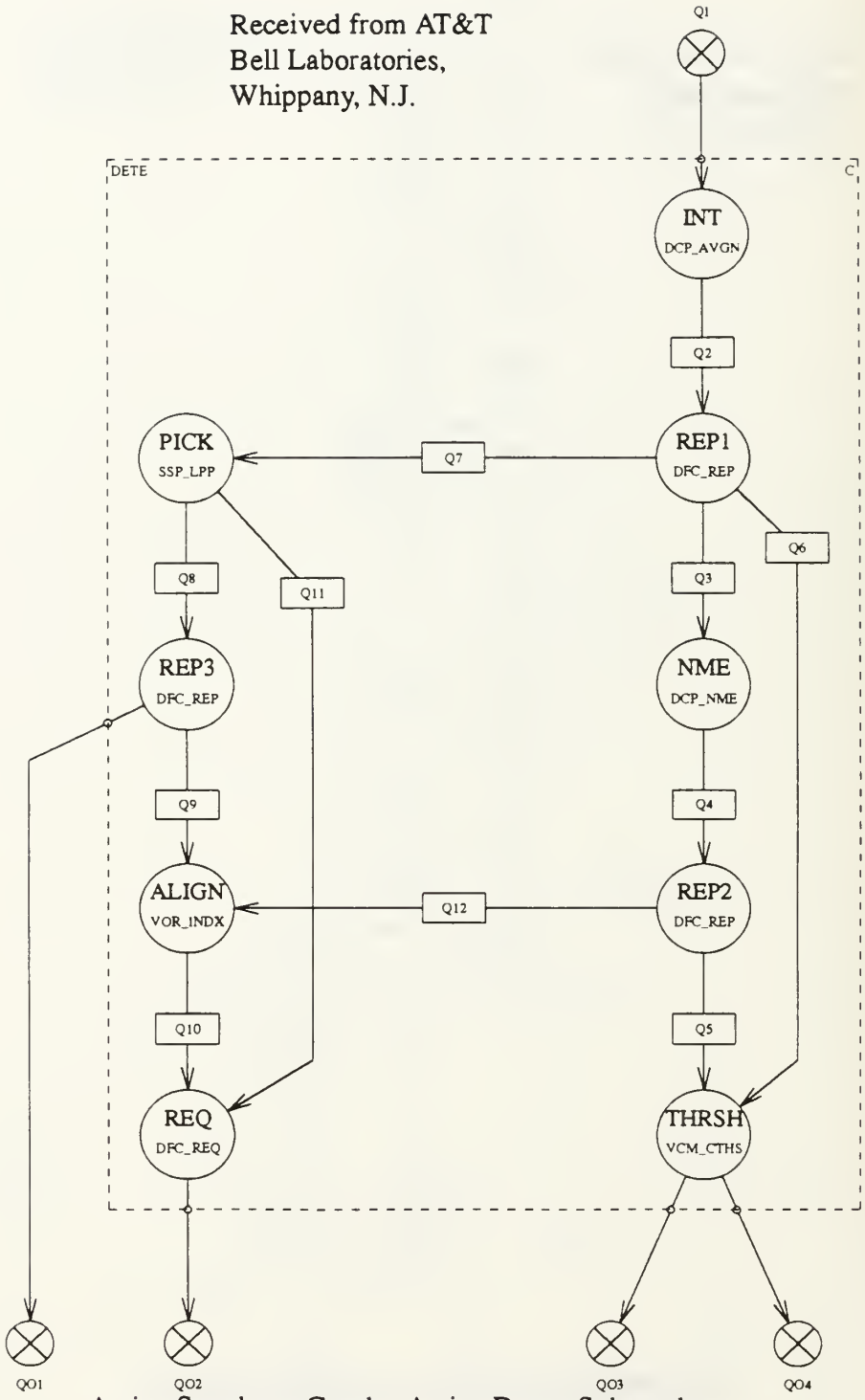
Active Sonobuoy Graph - Frequency Analysis Subgraph

Received from AT&T
Bell Laboratories,
Whippany, N.J.



Active Sonobouy Graph - Beamformer Subgraph

Received from AT&T
Bell Laboratories,
Whippany, N.J.



Active Sonobuoy Graph - Active Detect Subgraph

APPENDIX E: ACTIVE SONOBUOY GRAPH SPGN†

```
%%
%% rc_base.g: %Z% %P% %I% %G%
%%
%%**************************************************************%%
%% %%
%% GRAPH 'ACT_PROC ' %%
%% SPGN generated from GRED %%
%% Wed Apr 28 11:22:33 1993 %%
%% %%
%%**************************************************************%%
%GRAPH(ACT_PROC%% GRAPH SPGN FOR 'ACTIVE_PROCESS' OF
SONOBUOY BENCHMARK
 GIP = N: INT,%% INPUT BATCH SIZE
 NF: INT,%% NUMBER OF FREQUENCY CELLS
 NC: INT,%% NUMBER OF INPUT CHANNELS
 NB: INT,%% NUMBER OF BEAMS
 NB1: INT,%% NUMBER OF PEAK PICK BANDS
 NFFT: INT,%% FFT SIZE
 NAVG: INT,%% INTEGRATION TIME
 NW: INT,%% NUMBER OF FREQUENCY WEIGHTS
 NLEVEL: INT,%% NUMBER OF UANTIZATION LEVELS
 TS: INT,%% BANCSHIFT TABLE SIZE
 RATE: DFLOAT,%% INPUT SAMPLING RATE
 FC: DFLOAT,%% BANDSHIFT FREQUENCY
 RFFT: INT,%% FFT REDUNDANCY
 C: DFLOAT,%% CLIPPING CONSTANT
 R: DFLOAT,%% REPLACEMENT CONSTANT
 WIDTH: INT,%% WINDOW WIDTH
 TH: DFLOAT,%% DETECTION THRESHOLD
 IA: INT%% AVLV INDEX
 VAR = AVLV: INT ARRAY(8),%% LSR VALVE
 [1..NB]BWEIGHT: CFLOAT ARRAY(NF,NC)%% BEAMFORMING WEIGHTS
 INPUTQ = [1..NC]Q1: FIXED(0)%% FAMILY OF ACTIVE INPUTS
 OUTPUTQ = CELL: INT V_ARRAY(NF*NB),%% CONCATENATED BIN NUM-
BERS
```

† Created from original file received from AT&T Bell Laboratories, Whippany, N.J.

```
AMP: FLOAT V_ARRAY(NF*NB),%% CONCATENATED AMPLITUDES
LEVEL: INT,%% CONCATENATED LEVELS
INDEX: INT%% CONCATENATED INDICES
)


%% DECLARATIONS section (%GIP, %VAR, %QUEUE)

%GIP([1..NB]NB1C: INT
 INITIALIZE TO NB1)

%GIP(TVECTOR: FLOAT ARRAY(NLEVEL-1)%% QUANTIZATION THRESH-
OLDS
 INITIALIZE TO {1.E0,1.2E0,1.4E0,1.8E0,2.2E0,
 4.E0,6.E0})

%GIP(B: INT ARRAY(NF)%% BIN ADDRESS VECTOR
 INITIALIZE TO {512 OF 1})

%GIP([1..4]COEFF: DFLOAT ARRAY(5)%% IIR FILTER COEFFICIENTS
 INITIALIZE [1]COEFF TO {0.125E0,.1952965E0,.125E0,.362915E0,-.171509E0}
 INITIALIZE [2]COEFF TO {.25E0,.1297913E0,.25E0,.178711E0,-.736206E0}
 INITIALIZE [3]COEFF TO {0.125E0,.1952965E0,.125E0,.362915E0,-.171509E0}
 INITIALIZE [4]COEFF TO {.25E0,.1297913E0,.25E0,.178711E0,-.736206E0})

%GIP(FWEIGHT: FLOAT ARRAY(NF,NW)%% FREQUENCY WEIGHTS
 INITIALIZE TO {512*5 OF 1.0E0})

%GIP(N1: INT
 INITIALIZE TO 1024)

%GIP(N2: INT
 INITIALIZE TO 512)

%GIP(W: FLOAT ARRAY(NF,NW)
 INITIALIZE TO {NF*NW OF 1.0E0})

%VAR(R11: INT
 INITIALIZE TO 8)

%VAR(N11: INT
 INITIALIZE TO 1024)
```

```
%QUEUE(CNTR: INT%% FORMAT/PACK FEEDBACK COUNTER
 INITIALIZE TO 0,0,0)

%QUEUE([1..NC]Q2: CFLOAT)%% FAMILY OF NC SPECTRA
%QUEUE([1..NB]Q3: FLOAT)%% FAMILY OF BEAM MAGNITUDES
%QUEUE([1..NB]Q4: INT)%% FAMILY OF NB PEAK INDICES
%QUEUE([1..NB]Q7: INT V_ARRAY(NF))%% FAMILY OF NB BIN NUMBERS
%QUEUE([1..NB]Q5: INT)%% FAMILY OF NB PEAK LEVELS
%QUEUE([1..NB]Q6: FLOAT V_ARRAY(NF))%% FAMILY OF NB AMPLITUDES
%QUEUE([1..16]Q4A: CFLOAT)%% IIR FILTER OUT

%% TOPOLOGY section (%NODE, %SUBGRAPH)

%NODE(CONCAT4PRIMITIVE = DFC_VCAT
 PRIM_IN = NB,1,1,1,
 [1..NB]Q7THRESHOLD = 1
 PRIM_OUT = UNUSED,CELL)

%NODE(CONCAT3PRIMITIVE = DFC_VCAT
 PRIM_IN = NB,1,1,1,
 [1..NB]Q6THRESHOLD = 1
 PRIM_OUT = UNUSED,AMP)

%NODE(GRAMOUTPRIMITIVE = DFC_FMTPK
 PRIM_IN = NB1,NB,130,600,6,
 20,30,1,1,
 CNTRTHRESHOLD = 3,
 [1..NB]Q5THRESHOLD = NB1
 PRIM_OUT = CNTR,LEVEL)

%NODE(CONCAT1PRIMITIVE = DFC_CAT
 PRIM_IN = NB,1,[1..NB]NB1C,
 [1..NB]Q4THRESHOLD = NB1
 PRIM_OUT = INDEX)

%SUBGRAPH(BEAMFORMGRAPH = FREQ_BEAMFORM
 GIP = NF,NC,NB
 VAR = [1..NB]BWEIGHT
 INPUTQ = [1..NC]Q2
 OUTPUTQ = [1..NB]Q3)

%SUBGRAPH([I=1..NB]DETECTGRAPH = ACTIVE_DETECT
 GIP = NF,NAVG,NB1,NLEVEL,TVECTOR,
```

```
C,R,WIDTH,TH,IA
VAR = AVLV
INPUTQ = [I]Q3
OUTPUTQ = [I]Q4,[I]Q5,[I]Q6,[I]Q7)

%SUBGRAPH([I=1..NC/4]FTRANSFORMGRAPH = FREQ_TRANSFORM
GIP = N,TS,RATE,NFFT,NF,
NW,FWEIGHT
VAR = FC,[1..4]COEFF,RFFT,B
INPUTQ = [(I-1)*4+1..(I-1)*4+4]Q1
OUTPUTQ = [(I-1)*4+1..(I-1)*4+4]Q4A)

%NODE([I=1..NC]FQT2PRIMITIVE = SCH_FQT2
PIP_IN = R11,N11
PRIM_IN = N1,N1,N2,0,1,
B,NW,N2,W,
[I]Q4ATHRESHOLD = 1920VARIABLE CONSUME = 8*(N11/R11)
PRIM_OUT = [I]Q2)


%ENDGRAPH


%%*****************************************************************%%
%% %%
%% GRAPH 'FREQ_BEAMFORM ' %%
%% SPGN generated from GRED %%
%% Wed Apr 28 11:22:33 1993 %%
%% %%
%%*****************************************************************%%
%%
%%SPGN FOR 'FREQ_BEAMFORM' OF SONOBUOY BENCHMARK
%%CALLED BY ACTIVE_PROCESS
%%
%GRAPH(FREQ_BEAMFORM
GIP = NF: INT,%% NUMBER OF FREQUENCY BINS
NC: INT,%% NUMBER OF CHANNELS
NB: INT%% NUMBER OF BEAMS
VAR = [1..NB]W: CFLOAT ARRAY(NF,NC)%% BEAMFORMING WEIGHTS
INPUTQ = [1..NC]QI: CFLOAT%% FAMILY OF NC SPECTRA
OUTPUTQ = [1..NB]Q0: FLOAT%% FAMILY OF BEAM MAGNITUDES
)
```

```
%% DECLARATIONS section (%GIP, %VAR, %QUEUE)


%% TOPOLOGY section (%NODE, %SUBGRAPH)

%NODE(BEAMFORMERPRIMITIVE = SCH_FREQ
 PRIM_IN = NF,NC,NB,[1..NB]W,.961E0,
 .398E0,
 [1..NC]QITHRESHOLD = NF
 PRIM_OUT = [1..NB]Q0)


%ENDGRAPH


%%*****************************************************************%%
%% %%
%% GRAPH 'ACTIVE_DETECT ' %%
%% SPGN generated from GRED %%
%% Wed Apr 28 11:22:33 1993 %%
%% %%
%%*****************************************************************%%
%%
%%SPGN FOR 'ACTIVE_DETECT' OF SONOBUOY BENCHMARK
%%CALLED BY ACTIVE_PROCESS
%%
%GRAPH(ACTIVE_DETECT
 GIP = N: INT,%% BATCH SIZE
 NS: INT,%% AVERAGING CONSTANT
 NB: INT,%% NUMBER OF "OR" BANDS
 K: INT,%% NUMBER OF QUANTIZATION LEVELS
 TV: FLOAT ARRAY(K-1),%% QUANTIZATION THRESHOLDS
 C: DFLOAT,%% CLIPPING CONSTANT
 R: DFLOAT,%% REPLACEMENT CONSTANT
 W: INT,%% MEAN ESTIMATE WINDOW WIDTH
 T: DFLOAT,%% THRESHOLD
 IA: INT%% AVLV INDEX
 VAR = VLV: INT ARRAY(8)%% LSR VALVE
 INPUTQ = QI: FLOAT%% BEAM SPECTRAL MAGNITUDES
 OUTPUTQ = QO1: INT,%% OUTPUT PEAK INDICES
 QO2: INT,%% OUTPUT PEAK LEVELS
 QO3: FLOAT V_ARRAY(N),%% OUTPUT AMPLITUDES
```

```
 QO4: INT V_ARRAY(N)%% OUTPUT BIN NUMBERS
)


%% DECLARATIONS section (%GIP, %VAR, %QUEUE)


%% TOPOLOGY section (%NODE, %SUBGRAPH)

%NODE(DETECTPRIMITIVE = SCH_ACT
 PIP_IN = VLV(IA)
 PRIM_IN = N,NS,UNUSED,UNUSED,N/NB,
 TV,C,R,W,1,
 T,UNUSED,
 QITHRESHOLD = N*NS
 PRIM_OUT = UNUSED,UNUSED,QO1,
 QO2VARIABLE VALVE = VLV(IA),
 UNUSED,QO3,QO4)


%ENDGRAPH


%%************************************************************%%
%% %%
%% GRAPH 'FREQ_TRANSFORM ' %%
%% SPGN generated from GRED %%
%% Wed Apr 28 11:22:33 1993 %%
%% %%
%%************************************************************%%
%%
%%SPGN FOR 'FREQ_TRANSFORM' OF SONOBUOY BENCHMARK
%%CALLED BY ACTIVE_PROCESS
%%
%GRAPH(FREQ_TRANSFORM
 GIP = N: INT,%% BANDSHIFT/FILTER BATCH SIZE
 TS: INT,%% BANDSHIFT TABLE SIZE
 FS: DFLOAT,%% INPUT SAMPLING RATE
 N1: INT,%% FFT SIZE
 N2: INT,%% NUMBER OF OUTPUT BINS
 NW: INT,%% NUMBER OF WEIGHTS PER BIN
 W: FLOAT ARRAY(N2,NW)%% FREQUENCY WEIGHTS
 VAR = FC: DFLOAT,%% BANDSHIFT FREQUENCY
```

```
[1..4]COEFF: DFLOAT ARRAY(5),%% IIR FILTER COEFFICIENTS
R: INT,%% FFT REDUNDANCY
B: INT ARRAY(N2)%% BIN ADDRESS VECTOR
INPUTQ = [1..4]QI: FIXED(0)
OUTPUTQ = [1..4]Q40: CFLOAT
)


%% DECLARATIONS section (%GIP, %VAR, %QUEUE)

%QUEUE([1..4]Q2: INT
 INITIALIZE TO 0)

%QUEUE([1..4]QF_1: DCFLOAT
 INITIALIZE TO 4 OF <0.E0,0.E0>)

%QUEUE([1..4]QF_3: DCFLOAT
 INITIALIZE TO 4 OF <0.E0,0.E0>)

%QUEUE([1..4]QF_4: DCFLOAT
 INITIALIZE TO 4 OF <0.E0,0.E0>)

%QUEUE([1..4]QF_2: DCFLOAT
 INITIALIZE TO 4 OF <0.E0,0.E0>)


%% TOPOLOGY section (%NODE, %SUBGRAPH)

%NODE(FQT1PRIMITIVE = SCH_FQT1
 PRIM_IN = N,0,TS,FC,FS,
 [1..4]Q2THRESHOLD = 1,
 1,1,1,2,[1]COEFF,
 [2]COEFF,[3]COEFF,[4]COEFF,0,
 [1..4]QITHRESHOLD = N,
 [1..4]QF_1THRESHOLD = 4,
 [1..4]QF_2THRESHOLD = 4,
 [1..4]QF_3THRESHOLD = 4,
 [1..4]QF_4THRESHOLD = 4
 PRIM_OUT = [1..4]Q40,[1..4]Q2,[1..4]QF_1,[1..4]QF_2,[1..4]QF_3,
 [1..4]QF_4)

%ENDGRAPH
```

APPENDIX F: SONOBUOY GRAPH INPUT/OUTPUT FILE†

```
# The format of the i/o procedure file is tabular; the information
# content is simple, so this should cause no problems, and it reduces
# the amount of baggage in the simulator for interpreting this file.

# NAME is the symbolic identifier of the i/o procedure named in
# the %INITIO statement in the command program; it must match exactly;
# case is significant.
#
# TYPE is one of INPUT, BIDIRECTIONAL, or OUTPUT
#
# DATA RATES for graph inputs are expressed in words/second
#
# THRESH, READ, CONSUME, and PRODUCE amounts are in words
#
# comments look like this, everything on a line to the right
# of the sharp sign is discarded

#*****************************************************************
# COMMENTS COMMENTS COMMENTS COMMENTS COMMENTS COMMENTS COMMENTS
COMMENTS
#*****************************************************************

#THE FOLLOWING are the i/o procedures for the benchmark graph.

# The first section list the input and output procedures for the
# 50 instance of the passive graph. OMNI is the single input procedure and PGRAM??,
# PPEAK??, and PBIN??, is connected to each one of the 50 instances
# [where ?? represents the number of instance].

# The second section list the input and output procedures for a single
# instance of the passive graph. omni is the single input procedure
# and pgram, ppeak, pbin is connect the the last graph.

# The third and final section list the procedures of the active
# part of the benchmark graph. ACHAN is the 16 member input procedure
```

† Received from AT&T Bell Laboratories, Whippany, N.J.

and AINDEX, ALEVEL, AAMP, and ACELL are the four output procedures.

```
#*********************************************************************
# SECTION 1
#*********************************************************************

#ports
# nametypeoutput
#---------------
OMNIINPUT1

# rateproduce
#------------
 6503768

#ports
# nametypeinput
#---------------
PGRAM1OUTPUT1

#threshreadconsume
#-----------------
822822822
PGRAM2OUTPUT1

#threshreadconsume
#-----------------
822822822
PGRAM3OUTPUT1

#threshreadconsume
#-----------------
822822822
PGRAM4OUTPUT1



#threshreadconsume
#-----------------
408408408
PBIN49OUTPUT1

#threshreadconsume
```

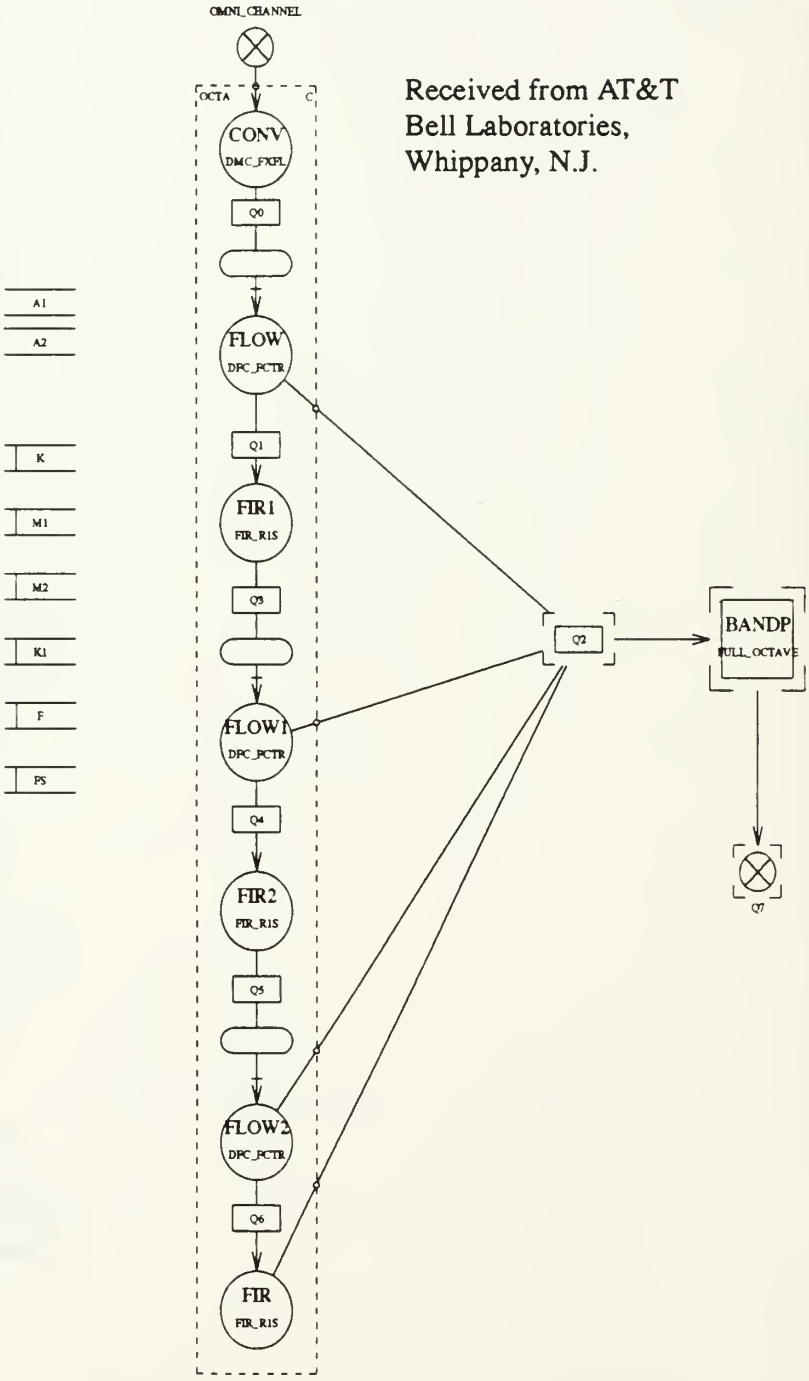
```
#-----------------
408408408

PBIN0OUTPUT1

#threshreadconsume
#-----------------
408408408

#ports
# nametypeinput
#---------------
PPEAK1OUTPUT1

#threshreadconsume
#-----------------
408408408
PPEAK2OUTPUT1


#threshreadconsume
#-----------------
408408408
PPEAK49OUTPUT1

#threshreadconsume
#-----------------
408408408

PPEAK0OUTPUT1

#threshreadconsume
#-----------------
408408408

#*****************************************************
# SECTION 2
#*****************************************************

#ports
# nametypeoutput
#---------------
omniINPUT1
```

```
# rateproduce
#------------
 6503768

#ports
# nametypeinput
#---------------
pgramOUTPUT1

#threshreadconsume
#-----------------
822822822

#ports
# nametypeinput
#---------------
pbinOUTPUT1

#threshreadconsume
#-----------------
408408408

#ports
# nametypeinput
#---------------
ppeak OUTPUT 1

#threshreadconsume
#-----------------
408408408

#*******************************************************
# SECTION 3
#*******************************************************
# Performance Envelope A

#ports
# nametypeoutput
#---------------
ACHANINPUT16

# rateproduce
```

```
#-----------
 67725679/60
 67725679/60
 67725679/60
 67725679/60
 67725679/60
 67725679/60
 67725679/60
 67725679/60
 67725679/60
 67725679/60
 67725679/60
 67725679/60
 67725679/60
 67725679/60
 67725679/60
 67725679/60

#ports
# nametypeinput
#----------------
ACELLOUTPUT1
#threshreadconsume
#------------------
819281928192

#ports
# nametypeinput
#----------------
AAMPOUTPUT1
#threshreadconsume
#------------------
819281928192

#ports
# nametypeinput
#----------------
ALEVELOUTPUT1
#threshreadconsume
#------------------
822822822

#ports
```

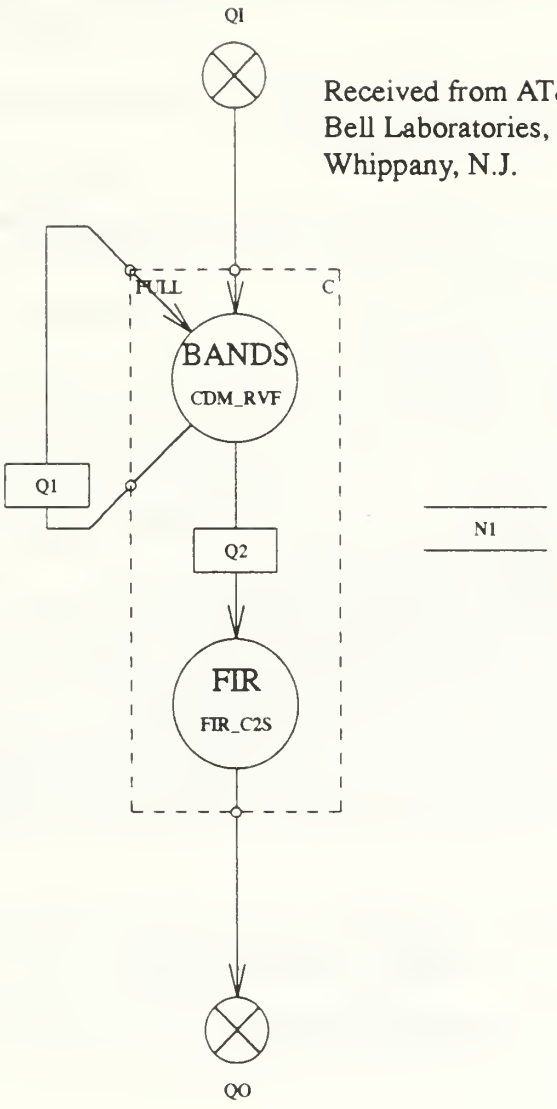
```
# nametypeinput
#---------------
AINDEXOUTPUT1
#threshreadconsume
#-----------------
512512512
```

APPENDIX G: PASSIVE (OCTAVE FILTER) GRAPH TOPOLOGY



Received from AT&T
Bell Laboratories,
Whippany, N.J.

Passive Sonobuoy Graph - Octave Filter Graph



Passive Sonobuoy Graph - Octave Filter - Full Octave

APPENDIX H: PASSIVE (OCTAVE FILTER) GRAPH SPGN†

```
%%
%% pass2.encode: %Z% %P% %I% %G%
%%
%%*************************************************************%%
%% %%
%% GRAPH 'OCTAVE_FILTER ' %%
%% SPGN generated from GRED %%
%% Mon May 3 15:03:06 1993 %%
%% %%
%%*************************************************************%%
%GRAPH(OCTAVE_FILTER
 GIP = N: INT,%% FILTER BATCH SIZE 8TH OCTAVE
 RATE: DFLOAT,%% INPUT DATA RATE
 FC: DFLOAT,%% OCTAVE 8 CENTER FREQUENCY
 NT1: INT,%% NUMBER OF INTEROCTAVE FILTER TAPS
 NT2: INT,%% NUMBER OF TAPS, FIRST STAGE
 NT3: INT%% NUMBER OF TAPS, SECOND STAGE
 VAR = [1..4]VLV: INT%% FULL OCTAVE OUTPUT VALVES
 INPUTQ = OMNI_CHANNEL: FIXED(0)%% OCTAVE FILTER INPUTS
 OUTPUTQ = [1..4]Q7: CFLOAT%% FAMILY OF OCTAVE BANDS
)


%% DECLARATIONS section (%GIP, %VAR, %QUEUE)

%GIP([1..4]K: INT%% OCTAVE CONSTANTS
 I=1..4
 INITIALIZE [I]K TO 2**(I-1))

%GIP([1..3]M1: INT%% INTEROCTAVE FILTER READ AMOUNT
 I=1..3
 INITIALIZE [I]M1 TO N/[I]K+(NT1-2))

%GIP([1..4]M2: INT%% FULL OCTAVE FILTER READ AMOUNT
 I=1..4 ·
```

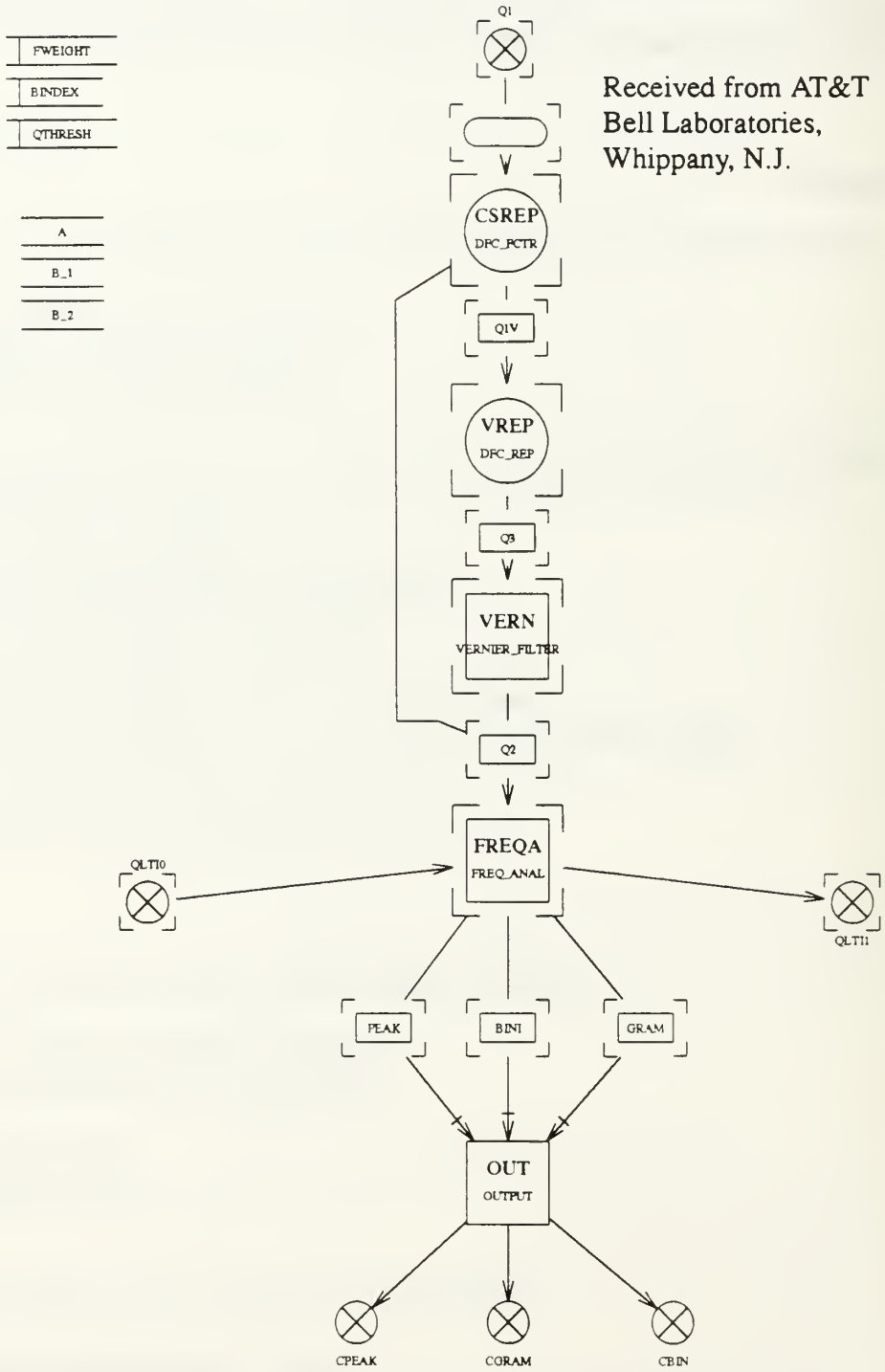
† Created from files received from AT&T Bell Laboratories, Whippany, N.J.

```
INITIALIZE [I]M2 TO N/[I]K)

%GIP([1..4]K1: FLOAT%% OCTAVE CONSTANTS
I=1..4
INITIALIZE [I]K1 TO 2.E0**(1-I))

%GIP([1..4]F: DFLOAT%% OCTAVE BANDSHIFT FREQUENCIES
I=1..4
INITIALIZE [I]F TO FC*[I]K1)

%GIP([1..4]FS: DFLOAT%% FULL OCTAVE FILTER INPUT RATES
I=1..4
INITIALIZE [I]FS TO RATE*[I]K1)

%GIP(A1: FLOAT ARRAY(NT1)%% INTEROCTAVE FILTER COEFFICIENTS
INITIALIZE TO {.00542537E0,0.E0,-.0170883E0,0.0E0,.03532707E0,
0.E0,-.0743693E0,0.0E0,.25E0,.399718E0,
.25E0,0.0E0,-.0743693E0,0.E0,.03532707E0,
0.E0,-.0170883E0,0.E0,.00542537E0})

%GIP(A2: FLOAT ARRAY(NT2+NT3)%% FULL OCTAVE FILTER COEFFICIENTS
INITIALIZE TO {-.0360940E0,0.E0,.25E0,.431116E0,.25E0,
0.E0,-.036094E0,.00542537E0,0.E0,-.0170883E0,
0.E0,.03532707E0,0.E0,-.0743693E0,0.E0,
.25E0,.399718E0,.25E0,0.E0,-.0743693E0,
0.E0,.03532707E0,0.E0,-.0170883E0,0.E0,
.00542537E0})

%QUEUE([1..4]Q2: FLOAT%% FAMILY OF BANDPASS OUTPUTS
INITIALIZE [1]Q2 TO (NT2-2)+(NT3-2)*2+1 OF 0.0E0
INITIALIZE [2]Q2 TO (NT2-2)+(NT3-2)*2+9 OF 0.0E0
INITIALIZE [3]Q2 TO (NT2-2)+(NT3-2)*2+13 OF 0.0E0
INITIALIZE [4]Q2 TO (NT2-2)+(NT3-2)*2+15 OF 0.0E0)


%% TOPOLOGY section (%NODE, %SUBGRAPH)

%SUBGRAPH([I=1..4]BANDPASS_FILTERGRAPH = FULL_OCTAVE
GIP = [I]M2,[I]F,[I]FS,NT2,NT3
VAR = A2,[I]VLV
INPUTQ = [I]Q2
OUTPUTQ = [I]Q7)
```

```
%NODE(OCTAVEPRIMITIVE = SCH_OCT
 PRIM_IN = N+119,NT1,NT1,NT1,2,
 2,2,A1,A1,A1,
 OMNI_CHANNELTHRESHOLD = N+119CONSUME = N
 PRIM_OUT = [1]Q2,[2]Q2,[3]Q2,[4]Q2)


%ENDGRAPH


%%*************************************************************%%
%% %%
%% GRAPH 'FULL_OCTAVE ' %%
%% SPGN generated from GRED %%
%% Mon May 3 15:03:06 1993 %%
%% %%
%%*************************************************************%%
%GRAPH(FULL_OCTAVE%% GRAPH SPGN FOR 'FULL_OCTAVE' OF
SONOBUOY BENCHMARK
 GIP = N: INT,%% BANDSHIFT BATCH SIZE
 F: DFLOAT,%% BANDSHIFT FREQUENCY
 FS: DFLOAT,%% INPUT DATA RATE
 NT1: INT,%% NUMBER OF TAPS, FIRST STAGE
 NT2: INT%% NUMBER OF TAPS, SECOND STAGE
 VAR = A: FLOAT ARRAY(NT1+NT2),%% FILTER COEFFICIENTS
 VLV: INT%% VALVE FOR FILTER OUTPUT QUEUE
 INPUTQ = QI: FLOAT%% FILTER INPUTS
 OUTPUTQ = QO: CFLOAT%% FILTER OUTPUTS
)


%% DECLARATIONS section (%GIP, %VAR, %QUEUE)

%GIP(N1: INT%% FIR READ AMOUNT
 INITIALIZE TO N+(NT1-2)+(NT2-2)*2)

%QUEUE(Q1: INT
 INITIALIZE TO 0)


%% TOPOLOGY section (%NODE, %SUBGRAPH)

%NODE(FULLPRIMITIVE = SCH_FUL
```

```
PIP_IN = VLV
PRIM_IN = N1,0,4096,F,FS,
Q1THRESHOLD = 1,
QITHRESHOLD = N1CONSUME = N,
NT1,NT2,2,2,A
PRIM_OUT = QOVARIABLE VALVE = VLV,
Q1)


%ENDGRAPH
```

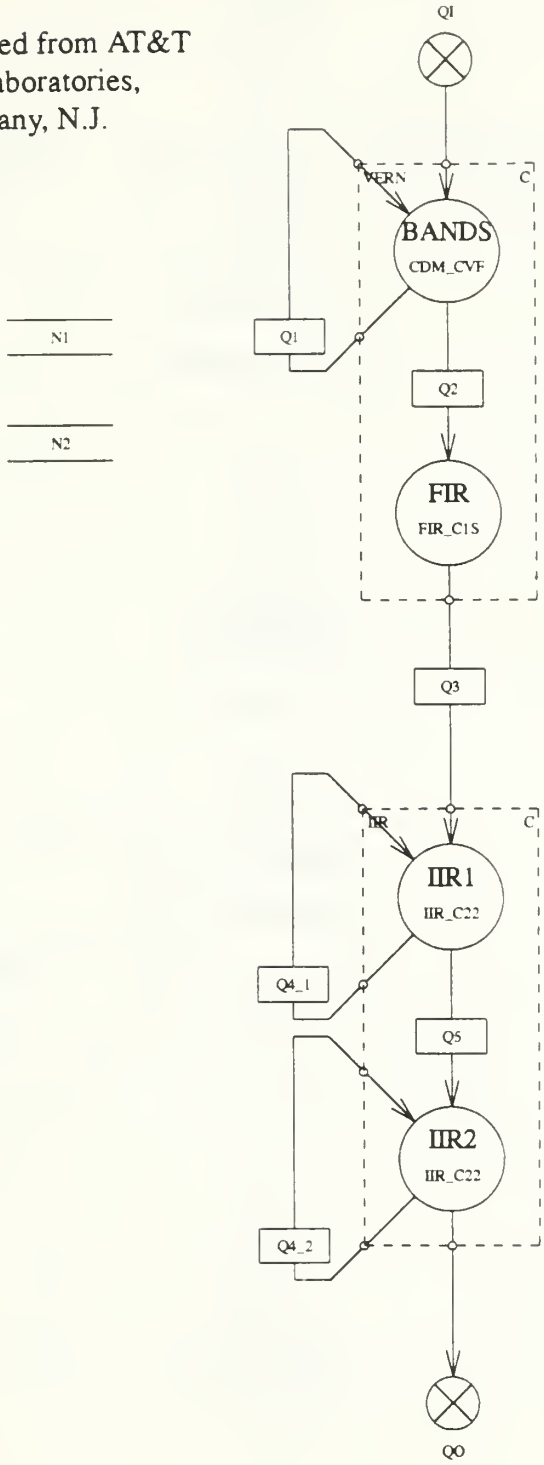
APPENDIX I: PASSIVE (FREQ ANALYSIS) GRAPH TOPOLOGY



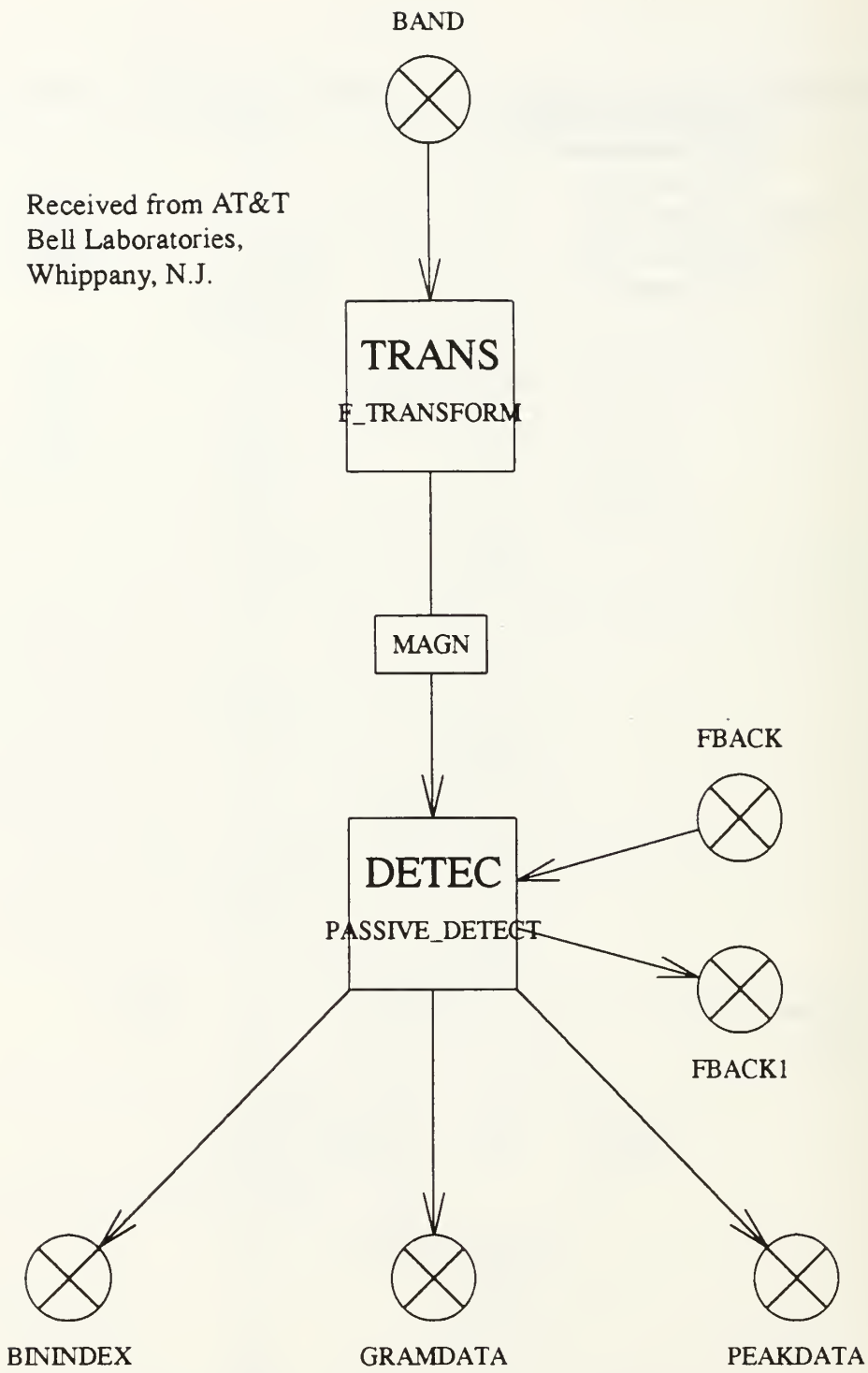
Received from AT&T
Bell Laboratories,
Whippany, N.J.

Passive Sonobuoy Graph - Frequency Analysis Root Level Graph

Received from AT&T
Bell Laboratories,
Whippany, N.J.

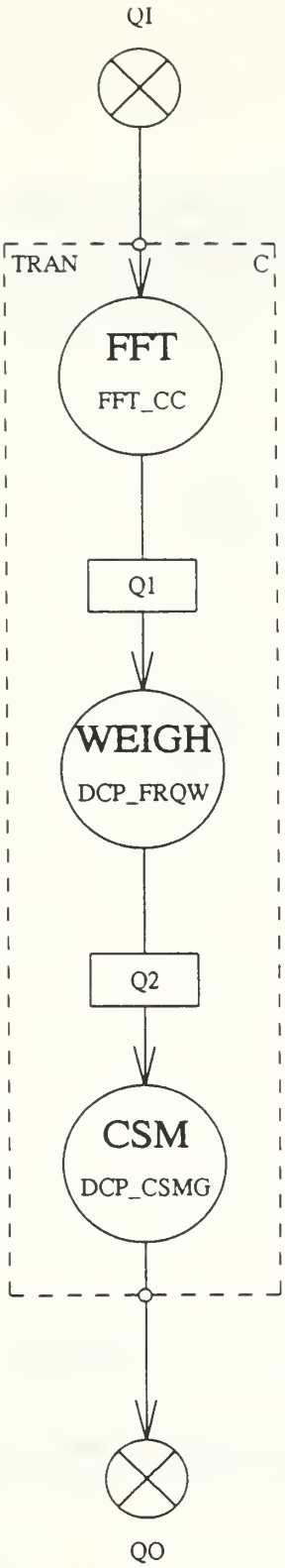


Passive Sonobuoy Graph - Frequency Analysis Graph - Vernier Filter Subgraph

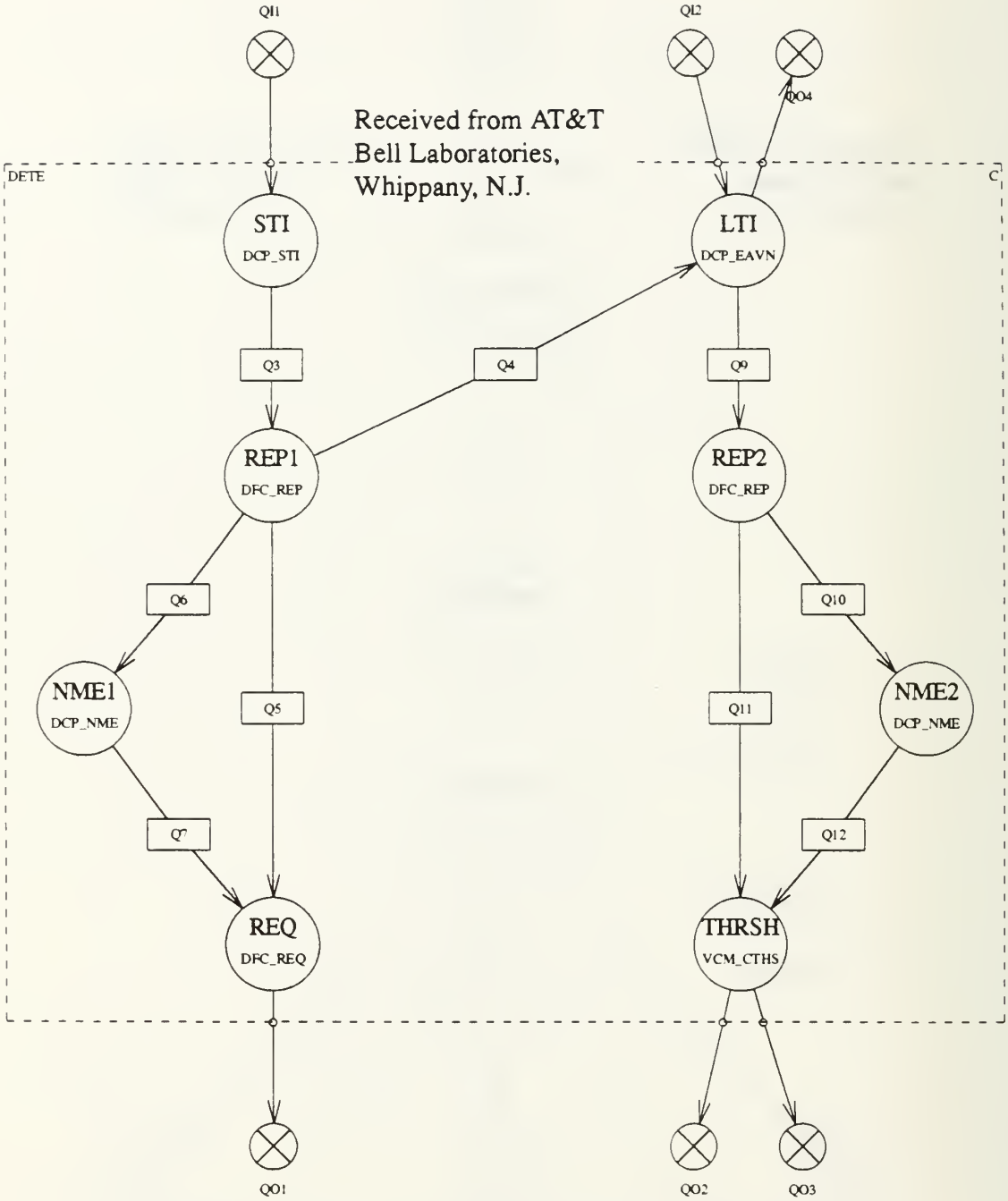


Passive Sonobuoy Graph - Frequency Analysis Graph - Frequency Analysis Subgraph

Received from AT&T
Bell Laboratories,
Whippany, N.J.

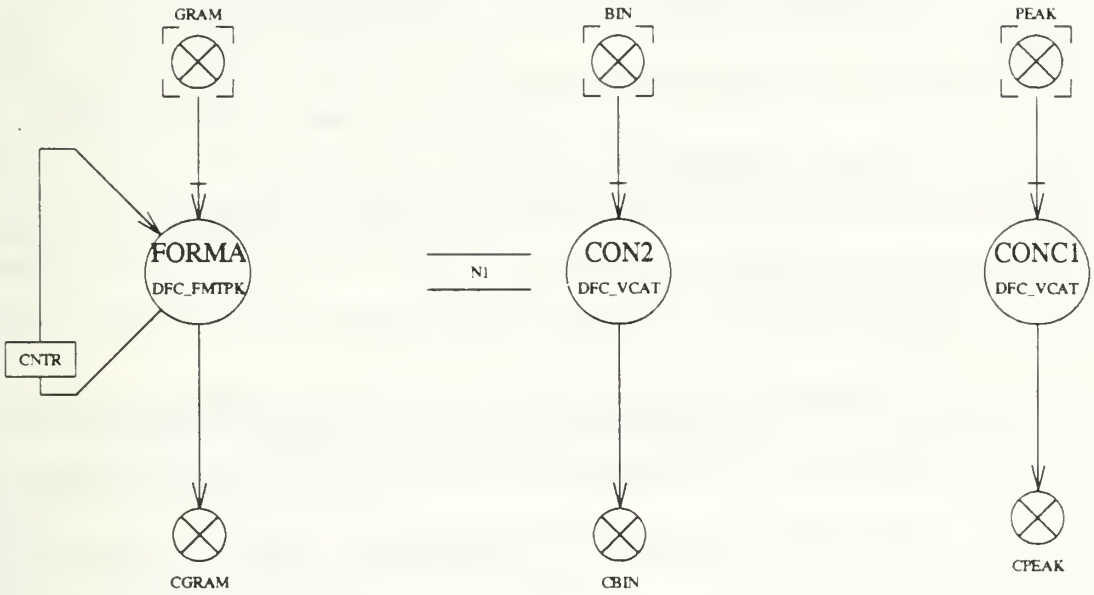


Frequency Analysis Subgraph - Frequency Transform Subgraph



Frequecy Analysis Subgraph - Passive Detect Subgraph

Received from AT&T
Bell Laboratories,
Whippany, N.J.



Passive Sonobuoy Graph - Frequency Analysis Graph - Output Subgraph

APPENDIX J: PASSIVE (FREQ ANALYSIS) GRAPH SPGN†

```
%%
%% pass1.encode: %Z% %P% %I% %G%
%%
%%************************************************************%%
%% %%
%% GRAPH 'FREQ_ANALYSIS ' %%%% SPGN generated from GRED %%
%% Mon May 3 14:59:01 1993 %%
%% %%
%%************************************************************%%
%%
%%SPGN FOR FREQ_ANALYSIS
%%USED IN PASSIVE PORTION OF SONOBUOY BENCHMARK
%%
%GRAPH(FREQ_ANALYSIS
 GIP = TNB: INT,%% TOTAL NUMBER OF PROCESSING BAND S (RANGE: 1 TO
8)
 NO: INT,%% NUMBER OF INPUT OCTAVES TO PROCESS (RANGE: 1 TO 4)
 NV: INT,%% TOTAL NUMBER OF VERNIER BANDS (RANGE: 1 TO 8)
 NOV: INT ARRAY(NO),%% NUMBER OF VERNIERS PER OCTAVE (RANGE: 1 TO
..)
 [1..4]FCTL_AMT: INT,%% NUMBER OF ELEMENTS TO PASS THROUGH FLOW
CONTROL
 FBS: INT ARRAY(NV),%% FILTER BATCH SIZE FOR VERNIER PROCESSING
 VO: INT ARRAY(NV),%% OCTAVE FOR VERNIER BAND (RANGE: 1 TO NO)
 VM: INT ARRAY(NV),%% MEMBER WITHIN OCTAVE (RANGE: 1 TO 8)
 FAI: INT ARRAY(TNB),%% FREQ_ANAL INPUT QUEUE INDEX
 FAIFB: INT ARRAY(TNB),%% LTI FEEDBACK QUEUE INDEX FOR FREQ_ANAL
 NT: INT,%% NUMBER OF TAPS FOR FIR FILTER IN VERNIER
 NFFT: INT,%% FFT SIZE
 NBIN: INT,%% NUMBER OF WEIGHTED BINS
 NWEIGHT: INT,%% NUMBER OF WEIGHTS/BIN
 NLEVEL: INT,%% NUMBER OF QUANTIZATION LEVELS
 [1..4]FLAG3: INT ARRAY(2),%% FLOW CONTROL INDEX VECTORS
 [1..8]FS: DFLOAT,%% SAMPLING RATE FOR BANDSHIFT IN VERNIER PRO-
CESSING
```

† Received from AT&T Bell Laboratories, Whippany, N.J.

```
[1..8]NSTI: INT,%% STI TIME CONSTANT
[1..8]NLTI: DFLOAT,%% LTI AVERAGE CONSTANT
[1..8]FGL: INT,%% LTI RESET FLAG
[1..8]K1: DFLOAT,%% CLIPPING CONSTANT
[1..8]L1: DFLOAT,%% STI REPLACEMENT CONSTANT
[1..8]W1: INT,%% STI MEAN ESTIMATE WINDOW WIDTH
[1..8]K2: DFLOAT,%% LTI CLIPPING CONSTANT
[1..8]L2: DFLOAT,%% LTI REPLACEMENT CONSTANT
[1..8]W2: INT,%% LTI MEAN ESTIMATE WINDOW
[1..8]DTHRESH: DFLOAT,%% LTI DETECTION THRESHOLD
NDTA: INT%% BATCH SIZE FOR GRAM DATA
VAR = RFFT: INT,%% FFT REDUNDANCY
[1..NV]F: DFLOAT,%% BANDSHIFT FREQUENCY FOR VERNIER
VSET: INT%% GRAM OUTPUT VALVE
INPUTQ = [1..NO]Q1: CFLOAT,%% OCTAVE INPUT DATA "J" = MEMBER INDEX
[1..TNB]QLTI0: DFLOAT%% LTI FEEDBACK
OUTPUTQ = [1..TNB]QLTI1: DFLOAT,%% LTI FEEDBACK QUEUE
CPEAK: FLOAT V_ARRAY(NDTA*TNB),%% CONCATENATED PEAK OUTPUTS
CGRAM: INT,%% GRAM OUTPUTS
CBIN: INT V_ARRAY(NDTA*TNB)%% CONCATENATED BIN OUTPUTS
)


%% DECLARATIONS section (%GIP, %VAR, %QUEUE)

%GIP(B_1: DFLOAT ARRAY(5)
%%
%%IIR FILTER COEFFICIENTS IN VERNIER
%%(1st OF CASCADE)
%%
 INITIALIZE TO {.0625E0,-.06651696E0,.0625E0,1.51126831E0,-.86311819E0})

%GIP([1..TNB]BINDEX: INT ARRAY(NBIN)%% BIN ADDRESS STARTING VEC-
TOR
 INITIALIZE TO
%% @(#) /b/cm/emsp/sef/sccs/c0/s1/s.s1.BINDEX.gd 1.4 5/11/89
{257, 258, 259, 260, 261, 262, 263, 264, 265,
266, 267, 268, 269, 270, 271, 272, 273,
274, 275, 276, 277, 278, 279, 280, 281,
282, 283, 284, 285, 286, 287, 288, 289,
290, 291, 292, 293, 294, 295, 296, 297,
298, 299, 300, 301, 302, 303, 304, 305,
306, 307, 308, 309, 310, 311, 312, 313,
```

314, 315, 316, 317, 318, 319, 320, 321,
322, 323, 324, 325, 326, 327, 328, 329,
330, 331, 332, 333, 334, 335, 336, 337,
338, 339, 340, 341, 342, 343, 344, 345,
346, 347, 348, 349, 350, 351, 352, 353,
354, 355, 356, 357, 358, 359, 360, 361,
362, 363, 364, 365, 366, 367, 368, 369,
370, 371, 372, 373, 374, 375, 376, 377,
378, 379, 380, 381, 382, 383, 384, 385,
386, 387, 388, 389, 390, 391, 392, 393,
394, 395, 396, 397, 398, 399, 400, 401,
402, 403, 404, 405, 406, 407, 408, 409,
410, 411, 412, 413, 414, 415, 416, 417,
418, 419, 420, 421, 422, 423, 424, 425,
426, 427, 428, 429, 430, 431, 432, 433,
434, 435, 436, 437, 438, 439, 440, 441,
442, 443, 444, 445, 446, 447, 448, 449,
450, 451, 452, 453, 454, 455, 456, 457,
458, 459, 460, 461, 462, 463, 464, 465,
466, 467, 468, 469, 470, 471, 472, 473,
474, 475, 476, 477, 478, 479, 480, 481,
482, 483, 484, 485, 486, 487, 488, 489,
490, 491, 492, 493, 494, 495, 496, 497,
498, 499, 500, 501, 502, 503, 504, 505,
506, 507, 508, 509, 510, 511, 512, 1,
2, 3, 4, 5, 6, 7, 8, 9,
10, 11, 12, 13, 14, 15, 16, 17,
18, 19, 20, 21, 22, 23, 24, 25,
26, 27, 28, 29, 30, 31, 32, 33,
34, 35, 36, 37, 38, 39, 40, 41,
42, 43, 44, 45, 46, 47, 48, 49,
50, 51, 52, 53, 54, 55, 56, 57,
58, 59, 60, 61, 62, 63, 64, 65,
66, 67, 68, 69, 70, 71, 72, 73,
74, 75, 76, 77, 78, 79, 80, 81,
82, 83, 84, 85, 86, 87, 88, 89,
90, 91, 92, 93, 94, 95, 96, 97,
98, 99, 100, 101, 102, 103, 104, 105,
106, 107, 108, 109, 110, 111, 112, 113,
114, 115, 116, 117, 118, 119, 120, 121,
122, 123, 124, 125, 126, 127, 128, 129,
130, 131, 132, 133, 134, 135, 136, 137,
138, 139, 140, 141, 142, 143, 144, 145,

```
146, 147, 148, 149, 150, 151, 152, 153,
154, 155, 156, 157, 158, 159, 160, 161,
162, 163, 164, 165, 166, 167, 168, 169,
170, 171, 172, 173, 174, 175, 176, 177,
178, 179, 180, 181, 182, 183, 184, 185,
186, 187, 188, 189, 190, 191, 192, 193,
194, 195, 196, 197, 198, 199, 200, 201,
202, 203, 204, 205, 206, 207, 208, 209,
210, 211, 212, 213, 214, 215, 216, 217,
218, 219, 220, 221, 222, 223, 224, 225,
226, 227, 228, 229, 230, 231, 232, 233,
234, 235, 236, 237, 238, 239, 240, 241,
242, 243, 244, 245, 246, 247, 248, 249,
250, 251, 252, 253, 254, 255, 256
}
)

%GIP(B_2: DFLOAT ARRAY(5)
%%
%%IIR FILTER COEFFICIENTS IN VERNIER
%%(2nd OF CASCADE)
%%
 INITIALIZE TO {.125E0,.05127928E0,.125E0,1.46595471E0,-.5917122E0})

%GIP([1..TNB]QTHRESH: DFLOAT ARRAY(NLEVEL-1)%% QUANTIZATION
THRESHOLD
 INITIALIZE TO {0.6E0,1.2E0,1.4E0,1.8E0,1.9E0,
 2.0E0,2.2E0})

%GIP(A: FLOAT ARRAY(7)%% FILTER COEFFICIENTS FOR VERNIER
 INITIALIZE TO {-
.035884509640E0,0.E0,.249186301952E0,.4289986601E0,.249186301952E0,
 0.E0,-.035884509640E0})

%GIP([1..TNB]FWEIGHT: FLOAT ARRAY(NBIN,NWEIGHT)%% FREQUENCY
WEIGHTS
 INITIALIZE TO
%% @(#) /b/cm/emsp/sef/sccs/c0/s1/s.s1.FFWGHT.gd 1.2 4/19/89
 {0.E0,-.25E0,.5E0,0.25E0,0.E0,0.E0,-.25E0,.5E0,0.25E0,0.E0,
  0.E0,-.25E0,.5E0,0.25E0,0.E0,0.E0,-.25E0,.5E0,0.25E0,0.E0}
)

%QUEUE([1..NO]Q1V: CFLOAT
```

```
INITIALIZE TO 5 OF <0.E0,0.E0>)

%QUEUE([1..NO,1..8]Q3: CFLOAT)%% INPUT FOR VERNIER_FILTER
%QUEUE([1..NV+NO]Q2: CFLOAT)%% INPUT FOR FREQ_ANAL SUBGRAPH
%QUEUE([1..TNB]GRAMDATA: INT)%% GRAM DATA
%QUEUE([1..TNB]BININDEX: INT V_ARRAY(NDTA))%% PEAK INDEX DATA
%QUEUE([1..TNB]PEAKDATA: FLOAT V_ARRAY(NBIN))%% PEAK DATA

%% TOPOLOGY section (%NODE, %SUBGRAPH)

%NODE([J=1..NO]CSREPPRIMITIVE = DFC_FCTR
 PRIM_IN = [J]FCTL_AMT,1,2,[J]FLAG3,
 FAMILY[[J]Q1]THRESHOLD = [J]FCTL_AMT
 PRIM_OUT = FAMILY[[J]Q1V,[NV+J]Q2])

%NODE([J=1..NO]VREPPRIMITIVE = DFC_REP
 PRIM_IN = 1,NOV(J),
 [J]Q1VTHRESHOLD = 1
 PRIM_OUT = FAMILY[[J,1..NOV(J)]Q3])

%SUBGRAPH([I=1..NV]VERN_FILTERGRAPH = VERNIER_FILTER
 GIP = FBS(I),NT
 VAR = [I]F,[VO(I)]FS,A,B_1,B_2
 INPUTQ = [VO(I),VM(I)]Q3
 OUTPUTQ = [I]Q2)

%SUBGRAPH(OUTGRAPH = OUTPUT
 GIP = TNB,NDTA
 INPUTQ = [1..TNB]GRAMDATA,[1..TNB]BININDEX,[1..TNB]PEAKDATA
 OUTPUTQ = CGRAM,CBIN,CPEAK)

%SUBGRAPH([I=1..TNB]FREQ_ANALGRAPH = FREQ_ANAL
 GIP = NFFT,NBIN,NWEIGHT,NLEVEL,[I]NSTI,
 [I]FWEIGHT
 VAR = RFFT,[I]BINDEX,[I]NLTI,[I]FGL,[I]K1,
 [I]L1,[I]W1,[I]K2,[I]L2,[I]W2,
 [I]QTHRESH,[I]DTHRESH,VSET
 INPUTQ = [FAI(I)]Q2,[FAIFB(I)]QLTI0
 OUTPUTQ = [I]BININDEX,[I]GRAMDATA,[I]PEAKDATA,[FAIFB(I)]QLTI1)


%ENDGRAPH
```

```
%%************************************************************%%
%% %%
%% GRAPH 'VERNIER_FILTER ' %%
%% SPGN generated from GRED %%
%% Mon May 3 14:59:01 1993 %%
%% %%
%%************************************************************%%
%%
%%SPGN FOR VERNIER FILTER
%%USED IN FREQ_ANALYSIS IN SONOBUOY BENCHMARK
%%
%GRAPH(VERNIER_FILTER
 GIP = N: INT,%% FILTER BATCH SIZE
 NT: INT%% NUMBER OF FIR TAPS
 VAR = F: DFLOAT,%% BANDSHIFT FREQUENCY
 FS: DFLOAT,%% INPUT SAMPLING RATE
 A: FLOAT ARRAY(NT),%% FIR FILTER COEFFICIENTS
 B_1: DFLOAT ARRAY(5),%% IIR FILTER COEFFICIENTS (1ST OF CASCADE)
 B_2: DFLOAT ARRAY(5)%% IIR FILTER COEFFICIENTS (2ND OF CASCADE)
 INPUTQ = QI: CFLOAT%% FILTER INPUTS
 %%
 OUTPUTQ = QO: CFLOAT%% FILTER OUTPUTS
)

%% DECLARATIONS section (%GIP, %VAR, %QUEUE)

%GIP(N1: INT%% FIR READ AMOUNT
 INITIALIZE TO (N+(NT-2)))

%GIP(N2: INT%% IIR READ AMOUNT
 INITIALIZE TO (N/2))

%QUEUE(Q1: INT%% PHASEPOINTER FEEDBACK
 INITIALIZE TO 0)

%QUEUE(Q3: CFLOAT)%% FIR FILTER OUTPUTS
%QUEUE(Q4_1: DCFLOAT%% IIR FILTER HISTORY
 INITIALIZE TO 4 OF <0.E0,0.E0>)

%QUEUE(Q4_2: DCFLOAT%% IIR FILTER HISTORY
 INITIALIZE TO 4 OF <0.E0,0.E0>)
```

```
%% TOPOLOGY section (%NODE, %SUBGRAPH)

%NODE(VERNPRIMITIVE = SCH_VRN
 PRIM_IN = N1,0,4096,F,FS,
 Q1THRESHOLD = 1,
 NT,2,A,
 QITHRESHOLD = N1CONSUME = N
 PRIM_OUT = Q3,Q1)

%NODE(IIRPRIMITIVE = SCH_IIR
 PRIM_IN = N2,2,2,B_1,B_2,
 0,
 Q3THRESHOLD = N2,
 Q4_1THRESHOLD = 4,
 Q4_2THRESHOLD = 4
 PRIM_OUT = QO,Q4_1,Q4_2)


%ENDGRAPH


%%*************************************************************%%
%% %%
%% GRAPH 'OUTPUT ' %%
%% SPGN generated from GRED %%
%% Mon May 3 14:59:01 1993 %%
%% %%
%%*************************************************************%%
%%
%%SPGN FOR OUTPUT
%%USED IN SONOBUOY BENCHMARK
%%
%GRAPH(OUTPUT
 GIP = NB: INT,%% NUMBER OF ANALYSIS
 N: INT%% BATCH SIZE FOR GRAM DATA
 INPUTQ = [1..NB]GRAM: INT,%% GRAM INPUTS
 [1..NB]BIN: INT V_ARRAY(N),%% BIN NUMBER INPUTS
 [1..NB]PEAK: FLOAT V_ARRAY(N)%% PEAK INPUTS
 OUTPUTQ = CGRAM: INT,%% CONCATENATED GRAM OUTPUTS
 CBIN: INT V_ARRAY(N*NB),%% CONCATENATED BIN OUTPUTS
 CPEAK: FLOAT V_ARRAY(N*NB)%% CONCATENATED PEAK OUTPUTS
```

```
)

%% DECLARATIONS section (%GIP, %VAR, %QUEUE)

%GIP(N1: INT%% V_ARRAY READ AMOUNT
 INITIALIZE TO 1)

%QUEUE(CNTR: INT%% FEEDBACK COUNTER
 INITIALIZE TO 0,0,0)


%% TOPOLOGY section (%NODE, %SUBGRAPH)

%NODE(FORMATPRIMITIVE = DFC_FMTPK
 PRIM_IN = 500,NB,10,600,6,
 20,10,1,4,
 CNTRTHRESHOLD = 3,
 [1..NB]GRAMTHRESHOLD = N
 PRIM_OUT = CNTR,CGRAM)

%NODE(CONCATENATE2PRIMITIVE = DFC_VCAT
 PRIM_IN = NB,1,1,1,
 [1..NB]BINTHRESHOLD = N1
 PRIM_OUT = UNUSED,CBIN)

%NODE(CONCATENATE1PRIMITIVE = DFC_VCAT
 PRIM_IN = NB,1,1,1,
 [1..NB]PEAKTHRESHOLD = N1
 PRIM_OUT = UNUSED,CPEAK)


%ENDGRAPH


%%**************************************************************%%
%% %%
%% GRAPH 'FREQ_ANAL ' %%
%% SPGN generated from GRED %%
%% Mon May 3 14:59:01 1993 %%
%% %%
%%**************************************************************%%
%%
```

```
%%SPGN FOR FREQ_ANAL SUBGRAPH
%%USED BY FREQ_ANALYSIS IN SONOBUOY BENCHMARK
%%
%GRAPH(FREQ_ANAL
 GIP = NFFT: INT,%% FFT SIZE
 NBIN: INT,%% NUMBER OF WEIGHTED BINS
 NWEIGHT: INT,%% NUMBER OF WEIGHTS PER BIN
 NLEVEL: INT,%% NUMBER OF QUANTIZATION LEVELS
 NSTI: INT,%% STI TIME CONSTANT
 FWEIGHT: FLOAT ARRAY(NBIN,NWEIGHT)%% FREQUENCY WEIGHTS
 VAR = RFFT: INT,%% FFT REDUNDANCY
 BINDEX: INT ARRAY(NBIN),%% BIN STARTING ADDRESS VECTOR
 NLTI: DFLOAT,%% LTI AVERAGING CONSTANT
 FG: INT,%% LTI RESET FLAG
 K1: DFLOAT,%% STI CLIPPING CONSTNAT
 L1: DFLOAT,%% STI REPLACEMENT CONSTANT
 W1: INT,%% STI MEAN ESTIMATE WINDOW WIDTH
 K2: DFLOAT,%% LTI CLIPPING CONSTNAT
 L2: DFLOAT,%% LTI REPLACEMENT CONSTANT
 W2: INT,%% LTI MEAN ESTIMATE WINDOW WIDTH
 QTHRESH: DFLOAT ARRAY(NLEVEL-1),%% QUANTIZATION THRESHOLDS
 DTHRESH: DFLOAT,%% LTI DETECTION THRESHOLD
 VSET: INT%% OUTPUT VALVE
 INPUTQ = BAND: CFLOAT,%% OCTAVE/VERNIER BAND DATA
 FBACK: DFLOAT%% LTI HISTORY
 OUTPUTQ = BININDEX: INT V_ARRAY(NBIN),%% PEAK BIN NUMBERS
 GRAMDATA: INT,%% QUANTIZED GRAM DATA
 PEAKDATA: FLOAT V_ARRAY(NBIN),%% LTI DETECTED PEAKS
 FBACK1: DFLOAT%% LTI FEEDBACK OUTPUT
)


%% DECLARATIONS section (%GIP, %VAR, %QUEUE)

%QUEUE(MAGNITUDE: FLOAT)%% SPECTRAL MAGNITUDES

%% TOPOLOGY section (%NODE, %SUBGRAPH)

%SUBGRAPH(TRANSFORMGRAPH = F_TRANSFORM
 GIP = NFFT,NBIN,NWEIGHT,FWEIGHT
 VAR = RFFT,BINDEX
 INPUTQ = BAND
 OUTPUTQ = MAGNITUDE)
```

```
%SUBGRAPH(DETECTGRAPH = PASSIVE_DETECT
 GIP = NBIN,NSTI,NLEVEL
 VAR = NSTI,NLTI,FG,K1,L1,
 W1,K2,L2,W2,QTHRESH,
 DTHRESH,VSET
 INPUTQ = MAGNITUDE,FBACK
 OUTPUTQ = FBACK1,GRAMDATA,PEAKDATA,BININDEX)


%ENDGRAPH


%%*************************************************************%%
%% %%
%% GRAPH 'F_TRANSFORM ' %%
%% SPGN generated from GRED %%
%% Mon May 3 14:59:01 1993 %%
%% %%
%%*************************************************************%%
%%
%%SPGN FOR F_TRANSFORM
%%USED BY FREQ_ANALYSIS IN SONOBUOY BENCHMARK
%%
%GRAPH(F_TRANSFORM
 GIP = N: INT,%% FFT SIZE
 M: INT,%% NUMBER OF WEIGHTED BINS
 NW: INT,%% NUMBER OF WEIGHTS PER BIN
 W: FLOAT ARRAY(M,NW)%% FREQUENCY WEIGHTS
 VAR = R: INT,%% FFT REDUNDANCY
 B: INT ARRAY(M)%% BIN ADDRESS VECTOR
 INPUTQ = QI: CFLOAT%% SPECTRAL ANALYSIS INPUTS
 OUTPUTQ = QO: FLOAT%% SPECTRAL MAGNITUDES
)


%% DECLARATIONS section (%GIP, %VAR, %QUEUE)


%% TOPOLOGY section (%NODE, %SUBGRAPH)

%NODE(TRANSPRIMITIVE = SCH_TRN
 PIP_IN = R
```

```
PRIM_IN = N,N,M,0,1,
B,NW,M,W,.961E0,
.398E0,
QITHRESHOLD = 2304VARIABLE CONSUME = 8*(N/R)
PRIM_OUT = QO)


%ENDGRAPH


%%************************************************************%%
%% %%
%% GRAPH 'PASSIVE_DETECT ' %%
%% SPGN generated from GRED %%
%% Mon May 3 14:59:01 1993 %%
%% %%
%%************************************************************%%
%%
%%SPGN FOR PASSIVE_DETECT
%%USED BY FREQ_ANALYSIS IN SONOBUOY BENCHMARK
%%
%GRAPH(PASSIVE_DETECT
 GIP = N: INT,%% BATCH SIZE
 M: INT,%% TIME SAMPLES PER STI EXECUTION
 K: INT%% NUMBER OF QUANTIZATION LEVELS
 VAR = CL: INT,%% STI TIME CONSTANT
 A: DFLOAT,%% LTI COEFFICIENT
 FG: INT,%% LTI RESET FLAG
 K1: DFLOAT,%% STI CLIPPING CONSTNAT
 L1: DFLOAT,%% STI REPLACEMENT CONSTANT
 W1: INT,%% STI MEANESTIMATE WINDOW WIDTH
 K2: DFLOAT,%% LTI CLIPPING CONSTNAT
 L2: DFLOAT,%% LTI REPLACEMENT CONSTNAT
 W2: INT,%% LTI MEAN ESTIMATE WINDOW WIDTH
 C: DFLOAT ARRAY(K-1),%% QUANTIZATION THRESHOLDS
 T: DFLOAT,%% LTI DETECTION THRESHOLD
 VSET: INT%% OUTPUT VALVE
 INPUTQ = QI1: FLOAT,%% SPECTRAL MAGNITUDES
 QI2: DFLOAT%% FEEDBACK
 OUTPUTQ = QO4: DFLOAT,%% FEEDBACK
 QO1: INT,%% OUTPUT PEAK INDICES
 QO2: FLOAT V_ARRAY(N),%% OUTPUT PEAK LEVELS
 QO3: INT V_ARRAY(N)%% OUTPUT AMPLITUDES
```

```
)

%% DECLARATIONS section (%GIP, %VAR, %QUEUE)

%% TOPOLOGY section (%NODE, %SUBGRAPH)

%NODE(DETECTPRIMITIVE = SCH_PASS
 PIP_IN = VSET
 PRIM_IN = N,M,CL,UNUSED,K1,
 L1,W1,C,A,FG,
 1,T,UNUSED,
 QI2THRESHOLD = N,
 QI1THRESHOLD = N*M
 PRIM_OUT = UNUSED,UNUSED,QO4,
 QO1VARIABLE VALVE = VSET,
 UNUSED,QO2,QO3)

%ENDGRAPH
```

APPENDIX K: PASSIVE GRAPH COMMAND PROGRAM†

```
#define TNB 8
#define NV 8
%INITCOMPROG()
int N8=8192, NT1 = 19, NT2 = 7, NT3 =19;
intNO=1;
intNOV[1], FCTL_AMT[4], FBS[8], VO[8], VM[8],
FAI[8], FAIFB[8],
NT=7, NFFT=512, NBIN=512, NWEIGHT=5, NLEVEL=8,
FLAG3[4][2];
intNSTI[8];
intFGL[8];
intW1[8];
intW2[8];
int NDTA=512;

inti, n, n2;

floatPRATE = 16.0e0;
floatFS[8];
floatf;

doubleFC=2.929860e0;
doubleNLTI[8];
doubleK1[8], L1[8];
doubleK2[8], L2[8];
doubleDTHRESH[8];

GV_IDVLV[4];
GV_IDRFFT, F[NV], VSET[107];

QUEUE_ID QIP, OCT_OUT[4];
QUEUE_ID QLTI[TNB], CGRAM, CBIN, CPEAK;

GRAPH_IDG, G1;
```

---

† Received from AT&T Bell Laboratories, Whippany, N.J.

```
IO_PROC_IDiop1, iop2, iop3, iop4, iop5, iop6;


/*Initialize the OCTAVE_FILTER GIPs
Initialization done in the int, float and double declarations
*/

/*Create and Initialize the OCTAVE_FILTER VARs
*/

n2 = 1;
for (n=0; n < 4; n++) {
 if ( n != 0) n2 = 0;
 VLV[n] = %CREATEGV(INT, n2); }

/*Create the OCTAVE_FILTER QUEUEs
*/

for (n2=0; n2 < 4; n2++)
 OCT_OUT[n2] = %CREATEQ(CFLOAT);

/*Initialize the FREQ_ANALYSIS GIPs
Some Initialization are in the declarations.
*/
NOV[0]=8;

for(i=0;i<4;i++) FCTL_AMT[i]=4096;
for(i=0;i<8;i++) FBS[i]=4096;
for(i=0;i<8;i++) VO[i]=1;
for(i=0;i<8;i++) VM[i]=i+1;
for(i=0;i<8;i++) FAI[i]=i+1;
for(i=0;i<8;i++) FAIFB[i]=i+1;

for(n=0;n<4;n++){
FLAG3[n][0]=1;
FLAG3[n][1]=0;}
for(n=0;n<8;n++) FS[n]=PRATE/4.0e0;
for(i=0;i<8;i++) NSTI[i]=8;
for(n=0;n<8;n++) NLTI[n]=1.25e-1;
for(i=0;i<8;i++) FGL[i]=0;
for(n=0;n<8;n++) K1[n]=2.5e0;
for(n=0;n<8;n++) L1[n]=1.5e0;
for(i=0;i<8;i++) W1[i]=32;
```

```
for(n=0;n<8;n++) K2[n]=2.0e;
for(n=0;n<8;n++) L2[n]=1.3e0;
for(i=0;i<8;i++) W2[i]=32;
for(n=0;n<8;n++) DTHRESH[n]=2.0e0;

/*Create and Initialize the FREQ_ANALYSIS VARs
*/
n2=2;
RFFT = %CREATEGV(INT, n2);
for(n2=0; n2 < NV; n2++){
 switch (n2 ){
 case 0: f=0.1e0;
 break;
 case 1: f=0.35e0;
 break;
 case 2: f=0.6e0;
 break;
 case 3: f=0.85e0;
 break;
 case 4: f=1.1e0;
 break;
 case 5: f=1.35e0;
 break;
 case 6: f=1.6e0;
 break;
 case 7: f=1.85e0;
 break;
 }
 F[n2]=%CREATEGV(DFLOAT, f);}
n2=1;
for(n=1;n<107;n++)
VSET[n] = %CREATEGV(INT, n2);

/*initialize the QUEUEs for the FREQ_ANALYSIS graph
*/
for (n2=0; n2 < TNB ; n2++) {
 QLTI[n2] = %CREATEQ(DFLOAT);
 %INITQ(QLTI[n2],512); }

/*
QUEUES attached to the I/O procedures (outside)
*/
QIP = %CREATEQ(FIXED);
```

```
CGRAM = %CREATEQ(INT);
CBIN = %CREATEQ(INT V_ARRAY(4096));
CPEAK = %CREATEQ(FLOAT V_ARRAY(4096));


/*%START this graphs
*/

 G = %START(OCTAVE_FILTER
GIP = N8, PRATE, FC, NT1, NT2, NT3
VAR = FAMILY(VLV[0], VLV[1], VLV[2], VLV[3])
INPUTQ = QIP
OUTPUTQ = FAMILY(OCT_OUT[0], OCT_OUT[1],
 OCT_OUT[2], OCT_OUT[3])
PRIORITY = 2);

 G1 = %START(FREQ_ANALYSIS
GIP = TNB,
NO,
NV,
NOV,
FAMILY(FCTL_AMT[0], FCTL_AMT[1],
 FCTL_AMT[2], FCTL_AMT[3]),
FBS,
VO,
VM,
FAI,
FAIFB,
NT,
NFFT,
NBIN,
NWEIGHT,
NLEVEL,
FAMILY(FLAG3[0], FLAG3[1], FLAG3[2], FLAG3[3]),
FAMILY(FS[0], FS[1], FS[2], FS[3],
 FS[4], FS[5], FS[6], FS[7]),
FAMILY(NSTI[0], NSTI[1], NSTI[2], NSTI[3],
 NSTI[4], NSTI[5], NSTI[6], NSTI[7]),
FAMILY(NLTI[0], NLTI[1], NLTI[2], NLTI[3],
 NLTI[4], NLTI[5], NLTI[6], NLTI[7]),
FAMILY(FGL[0], FGL[1], FGL[2], FGL[3],
 FGL[4], FGL[5], FGL[6], FGL[7]),
FAMILY(K1[0], K1[1], K1[2], K1[3],
```

```
 K1[4], K1[5], K1[6], K1[7]),
FAMILY(L1[0], L1[1], L1[2], L1[3],
 L1[4], L1[5], L1[6], L1[7]),
FAMILY(W1[0], W1[1], W1[2], W1[3],
 W1[4], W1[5], W1[6], W1[7]),
FAMILY(K2[0], K2[1], K2[2], K2[3],
 K2[4], K2[5], K2[6], K2[7]),
FAMILY(L2[0], L2[1], L2[2], L2[3],
 L2[4], L2[5], L2[6], L2[7]),
FAMILY(W2[0], W2[1], W2[2], W2[3],
 W2[4], W2[5], W2[6], W2[7]),
FAMILY(DTHRESH[0], DTHRESH[1], DTHRESH[2],
 DTHRESH[3], DTHRESH[4], DTHRESH[5],
 DTHRESH[6], DTHRESH[7]),
NDTA

VAR = RFFT,
FAMILY(F[0], F[1], F[2], F[3],
 F[4], F[5], F[6], F[7]),
VSET[1]

INPUTQ = FAMILY(OCT_OUT[0]),
FAMILY(QLTI[0], QLTI[1],
 QLTI[2], QLTI[3],
 QLTI[4], QLTI[5],
 QLTI[6], QLTI[7])

OUTPUTQ = FAMILY(QLTI[0], QLTI[1],
 QLTI[2], QLTI[3],
 QLTI[4], QLTI[5],
 QLTI[6], QLTI[7]),
CPEAK,
CGRAM,
CBIN
PRIORITY = 2);

if (%SCODE) exit(1);
iop1 = %INITIO(omni(1)INPUTQ = QIP);
iop2 = %INITIO(pgramOUTPUTQ = CGRAM);
iop3 = %INITIO(pbinOUTPUTQ = CBIN);
iop4 = %INITIO(ppeakOUTPUTQ = CPEAK);
%STARTIO(iop2);
%STARTIO(iop3);
```

```
%STARTIO(iop4);
%STARTIO(iop1);
%PRINT(%TERM, 2, G, G1);
%ENDPROGRAM
```

APPENDIX L: SELECTED FILES FOR RESTRUCTURING PROGRAMS

Graph.dat for the Correlator Graph

```
30
1 0 0 1 512 0 512 4096
2 1 0 2 512 0 512 4096
3 1 1 3 512 0 512 4096
4 1 3 4 512 0 512 4096
5 1 2 5 512 0 512 4096
6 1 5 6 512 0 512 4096
7 1 6 7 512 0 512 4096
8 1 7 8 512 0 512 4096
9 1 4 9 512 0 512 4096
10 1 9 10 512 0 512 4096
11 1 8 11 512 0 512 4096
12 1 11 12 512 0 512 4096
13 1 10 13 512 0 512 4096
14 1 12 14 512 0 512 4096
15 1 13 14 512 0 512 4096
16 1 13 15 512 0 512 4096
17 1 12 16 4096 0 4096 8192
18 1 14 17 4096 0 4096 8192
19 1 17 18 4096 0 4096 8192
20 1 15 19 4096 0 4096 8192
21 1 16 19 4096 0 4096 8192
22 1 19 20 4096 0 4096 8192
23 1 20 23 4096 0 4096 8192
25 1 18 23 4096 0 4096 8192
27 1 23 24 4096 0 4096 8192
28 1 24 25 4096 0 4096 8192
29 1 24 26 4096 0 4096 8192
30 1 26 27 4096 0 4096 8192
31 1 25 28 4096 0 4096 8192
32 1 27 28 4096 0 4096 8192

31 0 17 0 1 8 1 100
32 0 17 1 1 8 1 100
```

33 0 17 5 1 6 1 100
34 0 5 13 5 0 1 100
35 0 5 15 1 0 1 100
36 0 9 19 5 0 1 100
37 0 13 18 4 0 1 100
38 0 17 2 1 8 1 100
39 0 17 6 1 5 1 100
40 0 2 9 4 0 1 100
41 0 6 15 5 0 1 100
42 0 15 24 5 0 1 100
43 0 15 25 5 0 1 100
44 0 15 26 5 0 1 100
45 0 25 27 2 0 1 100
46 0 17 3 1 7 1 100
47 0 17 7 1 4 1 100
48 0 7 10 2 0 1 100
49 0 5 12 6 0 1 100
50 0 12 16 1 0 1 100
51 0 9 20 6 0 1 100
52 0 17 4 1 6 1 100
53 0 17 8 1 3 1 100
54 0 4 14 6 0 1 100
55 0 14 17 1 0 1 100
56 0 9 23 7 0 1 100

Cyl.dat for the Correlator Graph†

0 0 0 0 1 0 0 8650 5 -1 2180 13900 13 -5 16080 23302 11 -5 13900 18322 19 -7 24752 36178 18 -8 23302 31858

2 0 0 11410 6 -2 2180 19920 9 -3 11410 24190 15 -6 19920 27460 24 -10 27460 27460 25 -10 27460 28786 26 -10 27460 29332 27 -11 28786 30658

3 -1 0 11394 7 -3 3790 12894 10 -4 12894 17316 12 -6 15074 22606 16 -6 22606 28811 20 -8 24786 32071

4 -2 0 17740 8 -4 6560 21510 14 -7 17740 23110 17 -7 23110 37780 23 -9 25290 37780

28 2 1024 0 0 0 1 1

† This is part of the output of the restructurer programs [Ref. 16].

Appendix M: Restructured Correlator Graph SPGN†

```
%%
%% corr.rc.g: %Z% %P% %I% %G%
%%
%%**************************************************************%%
%% %%
%% GRAPH 'E006 ' %%
%% SPGN generated from GRED %%
%% Mon May 17 22:59:53 1993 %%
%% %%
%%**************************************************************%%
%GRAPH(E006
 GIP = SR: dfloat,
 F: dfloat,
 TC: dfloat,
 STI: int
 INPUTQ = [1..2]X: FIXED(2)
 OUTPUTQ = [1..2]Gramout: INT
)

%% DECLARATIONS section (%GIP, %VAR, %QUEUE)

%GIP(Lscan: INT%% 4 Long scan length
 INITIALIZE TO 16384)

%GIP(scan: INT%% 4 Short scan length
 INITIALIZE TO 4096)

%GIP(lags: INT%% number of correlation lags
 INITIALIZE TO 513)

%GIP(C1: DFLOAT ARRAY(3)
 INITIALIZE TO {3 OF 0.0E0})
```

† Generated from modified files received from AT&T Bell Laboratories, Whippany, N.J.

```
%GIP(C2: DFLOAT ARRAY(8)
 INITIALIZE TO {8 OF 0.0E0})

%GIP(Weights: FLOAT ARRAY(26)
 INITIALIZE TO {26 OF 1.0E0})

%GIP(D1: INT%% Decimations for filter
 INITIALIZE TO 2)

%GIP(D2: INT
 INITIALIZE TO 2)

%GIP(tap1: INT
 INITIALIZE TO 7)

%GIP(tap2: INT%% # of taps for filter
 INITIALIZE TO 19)

%GIP(filter_in: INT%% input to filter
 INITIALIZE TO D1*(D2*(scan-1)+tap2-1)+tap1)

%GIP(Appr1: FLOAT
 INITIALIZE TO 0.961E0)

%GIP(Appr2: FLOAT
 INITIALIZE TO 0.398E0)

%QUEUE(Fixflout1: DFLOAT)
%QUEUE(Band1out: DCFLOAT
 INITIALIZE TO 39 OF <0.0E0,0.0E0>)

%QUEUE(Band2out: DCFLOAT
 INITIALIZE TO 39 OF <0.0E0,0.0E0>)

%QUEUE(Fixflout2: DFLOAT)
%QUEUE(Fir2out: DCFLOAT)
%QUEUE(Fir1out: DCFLOAT)
%QUEUE(Zfilout: DCFLOAT)
%QUEUE(FFT2out: DCFLOAT)
%QUEUE(FFT1out: DCFLOAT)
%QUEUE(Wind1out: DCFLOAT)
%QUEUE(Wind2out: DCFLOAT)
%QUEUE(IFFTout: DCFLOAT)
```

```
%QUEUE(PwrMultout: DFLOAT)
%QUEUE(Sqrtout: DFLOAT)
%QUEUE(Asqrtout: DFLOAT ARRAY(1))
%QUEUE(Magout: DFLOAT)
%QUEUE(Divout: DFLOAT)
%QUEUE(STIout: DFLOAT)
%QUEUE(EAVNout: DFLOAT)
%QUEUE(Multout: DCFLOAT)
%QUEUE(Pwr1out: DFLOAT)
%QUEUE(Pwr2out: DFLOAT)
%QUEUE(Coeffptr1: INT
 INITIALIZE TO 1)

%QUEUE(Coeffptr2: INT
 INITIALIZE TO 1)

%QUEUE(EAVNfeed: DFLOAT
 INITIALIZE TO lags OF 0.0E0)

%QUEUE(refvect2: DFLOAT
 INITIALIZE TO lags OF 1.0E0)

%QUEUE([1..2]Rep2out: DCFLOAT)
%QUEUE([1..2]Rep1out: DCFLOAT)
%QUEUE([1..2]Rep3out: DFLOAT)
%QUEUE(refvect: DFLOAT
 INITIALIZE TO lags OF 1.0E0)

%QUEUE([1..8]RC1: TRIGGER
 INITIALIZE [1..2]RC1 TO 8
 INITIALIZE [3..3]RC1 TO 7
 INITIALIZE [4..5]RC1 TO 6
 INITIALIZE [6..6]RC1 TO 5
 INITIALIZE [7..7]RC1 TO 4
 INITIALIZE [8..8]RC1 TO 3)

%QUEUE([1..3]RC2: TRIGGER
 INITIALIZE [1..3]RC2 TO 0)

%QUEUE([1..2]RC3: TRIGGER
 INITIALIZE [1..2]RC3 TO 0)

%QUEUE(RC4: TRIGGER)
```

```
%QUEUE(RC5: TRIGGER)
%QUEUE(RC6: TRIGGER)
%QUEUE(RC8: TRIGGER)
%QUEUE(RC9: TRIGGER)
%QUEUE([1..3]RC10: TRIGGER
 INITIALIZE [1..3]RC10 TO 0)

%QUEUE(RC11: TRIGGER)
%QUEUE(RC12: TRIGGER)
%QUEUE(RC13: TRIGGER)

%% TOPOLOGY section (%NODE, %SUBGRAPH)

%NODE(Fixfl1PRIMITIVE = DMC_FXFL
 PIP_IN = [1..1]RC1THRESHOLD = 1
 PRIM_IN = Lscan,
 [1]XTHRESHOLD = LscanREAD = LscanCONSUME = Lscan
 PRIM_OUT = Fixflout1)

%NODE(Fixfl2PRIMITIVE = DMC_FXFL
 PIP_IN = [2..2]RC1THRESHOLD = 1
 PRIM_IN = Lscan,
 [2]XTHRESHOLD = LscanREAD = LscanCONSUME = Lscan
 PRIM_OUT = Fixflout2
 PIP_OUT = RC5PULSE = 1)

%NODE(Band1PRIMITIVE = CDM_RVF
 PIP_IN = [3..3]RC1THRESHOLD = 1
 PRIM_IN = Lscan,Unused,1024,F,SR,
 Coeffptr1THRESHOLD = 1,
 Fixflout1THRESHOLD = LscanREAD = LscanCONSUME = Lscan
 PRIM_OUT = Band1out,Coeffptr1)

%NODE(FIR1PRIMITIVE = FIR_C2S
 PIP_IN = [4..4]RC1THRESHOLD = 1
 PRIM_IN = filter_in,tap1,tap2,D1,D2,
 Weights,
 Band1outTHRESHOLD = filter_inREAD = filter_inCONSUME = Lscan
 PRIM_OUT = Fir1out
 PIP_OUT = RC12PULSE = 1)

%NODE(Band2PRIMITIVE = CDM_RVF
 PIP_IN = [5..5]RC1THRESHOLD = 1
```

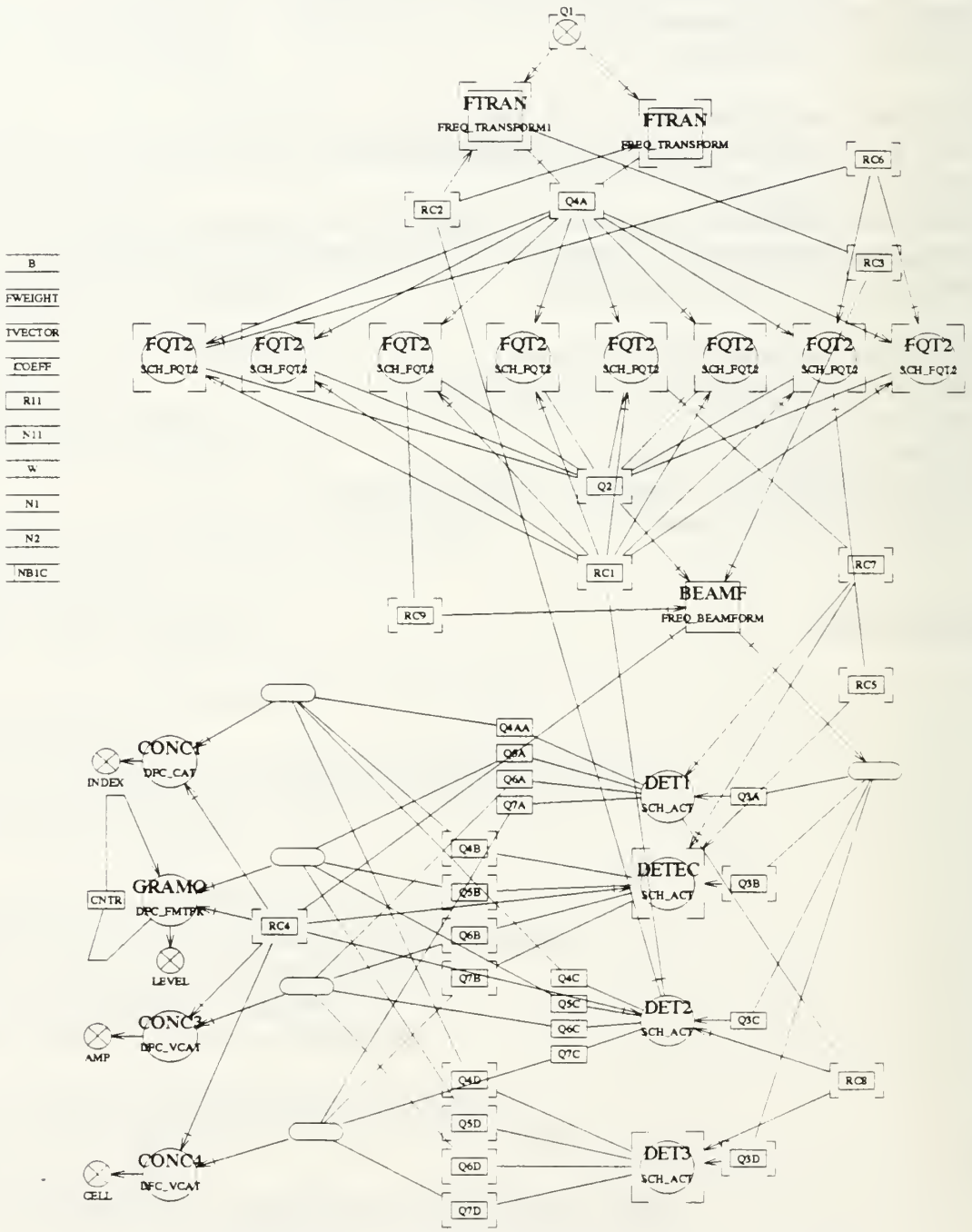
```
PRIM_IN = Lscan,Unused,1024,F,SR,
Coeffptr2THRESHOLD = 1,
Fixflout2THRESHOLD = LscanREAD = LscanCONSUME = Lscan
PRIM_OUT = Band2out,Coeffptr2
PIP_OUT = [1..3]RC2PULSE = 1)

%NODE(FIR2PRIMITIVE = FIR_C2S
PIP_IN = [6..6]RC1THRESHOLD = 1
PRIM_IN = filter_in,tap1,tap2,D1,D2,
Weights,
Band2outTHRESHOLD = filter_inREAD = filter_inCONSUME = Lscan
PRIM_OUT = Fir2out
PIP_OUT = RC6PULSE = 1)

%NODE(ZeroFillPRIMITIVE = DFC_REORD
PIP_IN = [7..7]RC1THRESHOLD = 1
PRIM_IN = scan,scan,3,2,256,
767,
Fir2outTHRESHOLD = scanREAD = scanCONSUME = scan
PRIM_OUT = Zfilout
PIP_OUT = RC8PULSE = 1)

%NODE(FFT2PRIMITIVE = FFT_CC
PIP_IN = [8..8]RC1THRESHOLD = 1
PRIM_IN = scan/4,scan/4,0,1,
ZfiloutTHRESHOLD = scanREAD = scanCONSUME = scan
PRIM_OUT = FFT2out)

%NODE(FFT1PRIMITIVE = FFT_CC
PIP_IN = RC5THRESHOLD = 4CONSUME = 1
PRIM_IN = scan/4,scan/4,0,1,
Fir1outTHRESHOLD = scanREAD = scanCONSUME = scan
PRIM_OUT = FFT1out
PIP_OUT = [1..2]RC3PULSE = 1)

%NODE(Window1PRIMITIVE = DCP_HAMN
PIP_IN = RC8THRESHOLD = 2CONSUME = 1
PRIM_IN = scan/4,1,
FFT1outTHRESHOLD = scanREAD = scanCONSUME = scan
PRIM_OUT = Wind1out)

%NODE(Window2PRIMITIVE = DCP_HAMN
PIP_IN = [3..3]RC2THRESHOLD = 5CONSUME = 1
```

```
 PRIM_IN = scan/4,1,
 FFT2outTHRESHOLD = scanREAD = scanCONSUME = scan
 PRIM_OUT = Wind2out)

%NODE(Replicate2PRIMITIVE = DFC_REP
 PIP_IN = [2..2]RC2THRESHOLD = 6CONSUME = 1
 PRIM_IN = SCAN,2,
 Wind2outTHRESHOLD = scanREAD = scanCONSUME = scan
 PRIM_OUT = [1..2]Rep2out
 PIP_OUT = RC9PULSE = 1)

%NODE(Replicate1PRIMITIVE = DFC_REP
 PIP_IN = [1..1]RC2THRESHOLD = 5CONSUME = 1
 PRIM_IN = SCAN,2,
 Wind1outTHRESHOLD = scanREAD = scanCONSUME = scan
 PRIM_OUT = [1..2]Rep1out
 PIP_OUT = RC4PULSE = 1)

%NODE(MultXYPRIMITIVE = VCC_VMUL
 PIP_IN = RC12THRESHOLD = 6CONSUME = 1
 PRIM_IN = scan,0,
 [1]Rep1outTHRESHOLD = scanREAD = scanCONSUME = scan,
 [1]Rep2outTHRESHOLD = scan
 PRIM_OUT = Multout
 PIP_OUT = RC13PULSE = 1)

%NODE(PowerXPRIMITIVE = VOC_PWR
 PIP_IN = RC6THRESHOLD = 5CONSUME = 1
 PRIM_IN = scan/4,
 [2]Rep1outTHRESHOLD = scanREAD = scanCONSUME = scan
 PRIM_OUT = Unused,Pwr1out
 PIP_OUT = [1..3]RC10PULSE = 1)

%NODE(PowerYPRIMITIVE = VOC_PWR
 PIP_IN = RC9THRESHOLD = 1
 PRIM_IN = scan/4,
 [2]Rep2outTHRESHOLD = scanREAD = scanCONSUME = scan
 PRIM_OUT = Unused,Pwr2out)

%NODE(InverseFFTPRIMITIVE = FFT_CC
 PIP_IN = RC13THRESHOLD = 1
 PRIM_IN = scan/4,lags,1,257,
 MultoutTHRESHOLD = scanREAD = scanCONSUME = scan
```

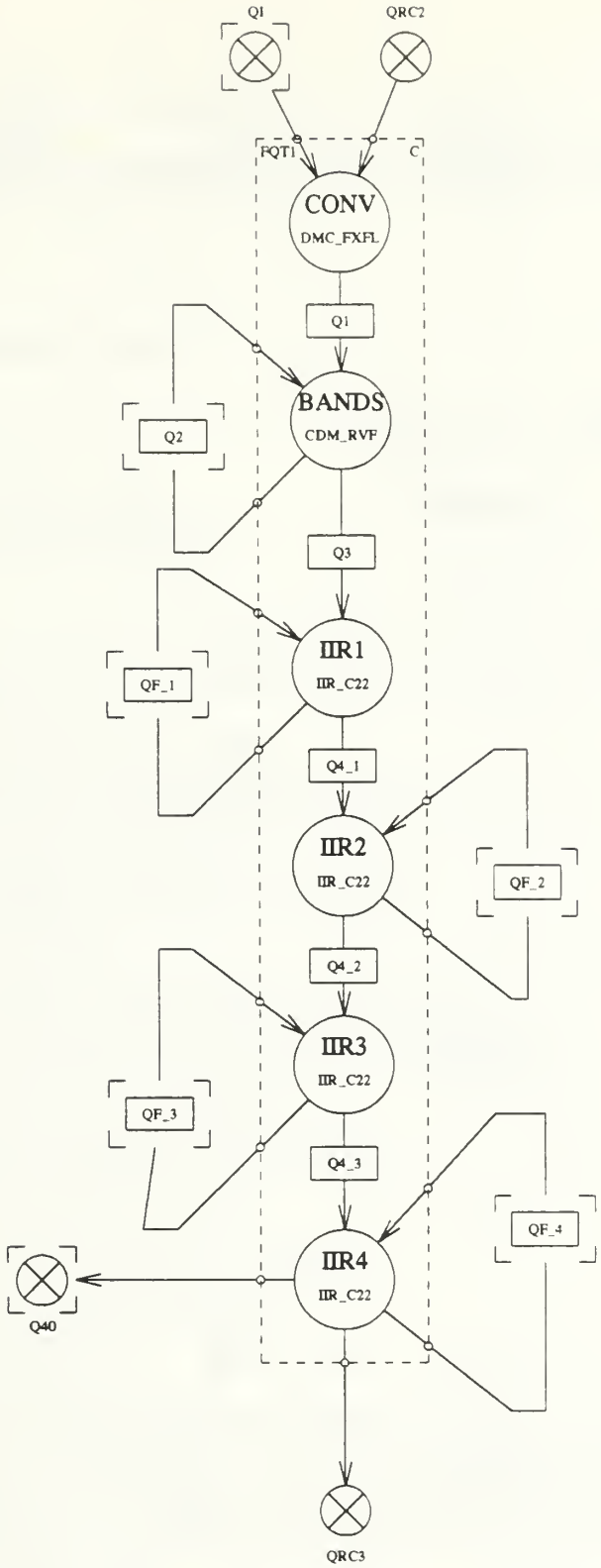
```
 PRIM_OUT = IFFTout
 PIP_OUT = [1..8]RC1PULSE = 1)

%NODE(MagnitudePRIMITIVE = DCP_CSMG
 PIP_IN = RC4THRESHOLD = 6CONSUME = 1
 PRIM_IN = 4*lags,Appr1,Appr2,
 IFFToutTHRESHOLD = 4*lagsREAD = 4*lagsCONSUME = 4*lags
 PRIM_OUT = Magout)

%NODE(MultPowerPRIMITIVE = VRR_VMUL
 PIP_IN = [2..2]RC3THRESHOLD = 5CONSUME = 1
 PRIM_IN = 4,
 Pwr2outTHRESHOLD = 4,
 Pwr1outTHRESHOLD = 4
 PRIM_OUT = PwrMultout)

%NODE(SqrtPRIMITIVE = VOR_VSQR
 PRIM_IN = 1,
 PwrMultoutTHRESHOLD = 1
 PRIM_OUT = Sqrtout)

%NODE(ChangePRIMITIVE = DMC_EMC
 PRIM_IN = 1,
 SqrtoutTHRESHOLD = 1
 PRIM_OUT = Asqrtout)

%NODE(NormalizePRIMITIVE = VRR_VDIV
 PRIM_IN = lags,
 MagoutTHRESHOLD = lagsREAD = lagsCONSUME = lags,
 AsqrtoutTHRESHOLD = 1READ = 1CONSUME = 1
 PRIM_OUT = Divout)

%NODE(IntegratePRIMITIVE = DCP_STI
 PIP_IN = [1..1]RC3THRESHOLD = 7CONSUME = 1
 PRIM_IN = lags,STI,STI,Unused,Unused,
 DivoutTHRESHOLD = lags*STI
 PRIM_OUT = Unused,Unused,STIout)

%NODE(Replicate3PRIMITIVE = DFC_REP
 PIP_IN = [1..1]RC10THRESHOLD = 5CONSUME = 1
 PRIM_IN = lags,2,
 STIoutTHRESHOLD = lagsREAD = lagsCONSUME = lags
 PRIM_OUT = [1..2]Rep3out)
```

```
%NODE(GramPRIMITIVE = DFC_REQ
 PIP_IN = [2..2]RC10THRESHOLD = 5CONSUME = 1
 PRIM_IN = lags,C2,
 [1]Rep3outTHRESHOLD = lagsREAD = lagsCONSUME = lags,
 refvectTHRESHOLD = lagsREAD = lagsCONSUME = 0
 PRIM_OUT = [1]Gramout
 PIP_OUT = RC11PULSE = 1)

%NODE(ExpAvgPRIMITIVE = DCP_EAVN
 PIP_IN = [3..3]RC10THRESHOLD = 5CONSUME = 1
 PRIM_IN = 1,lags,TC,0,
 EAVNfeedTHRESHOLD = lagsREAD = lagsCONSUME = lags,
 [2]Rep3outTHRESHOLD = lagsREAD = lagsCONSUME = lags
 PRIM_OUT = EAVNout,EAVNfeed)

%NODE(AscanPRIMITIVE = DFC_REQ
 PIP_IN = RC11THRESHOLD = 2CONSUME = 1
 PRIM_IN = lags,C2,
 EAVNoutTHRESHOLD = lags,
 refvect2THRESHOLD = lagsREAD = lagsCONSUME = 0
 PRIM_OUT = [2]Gramout)


%ENDGRAPH
```

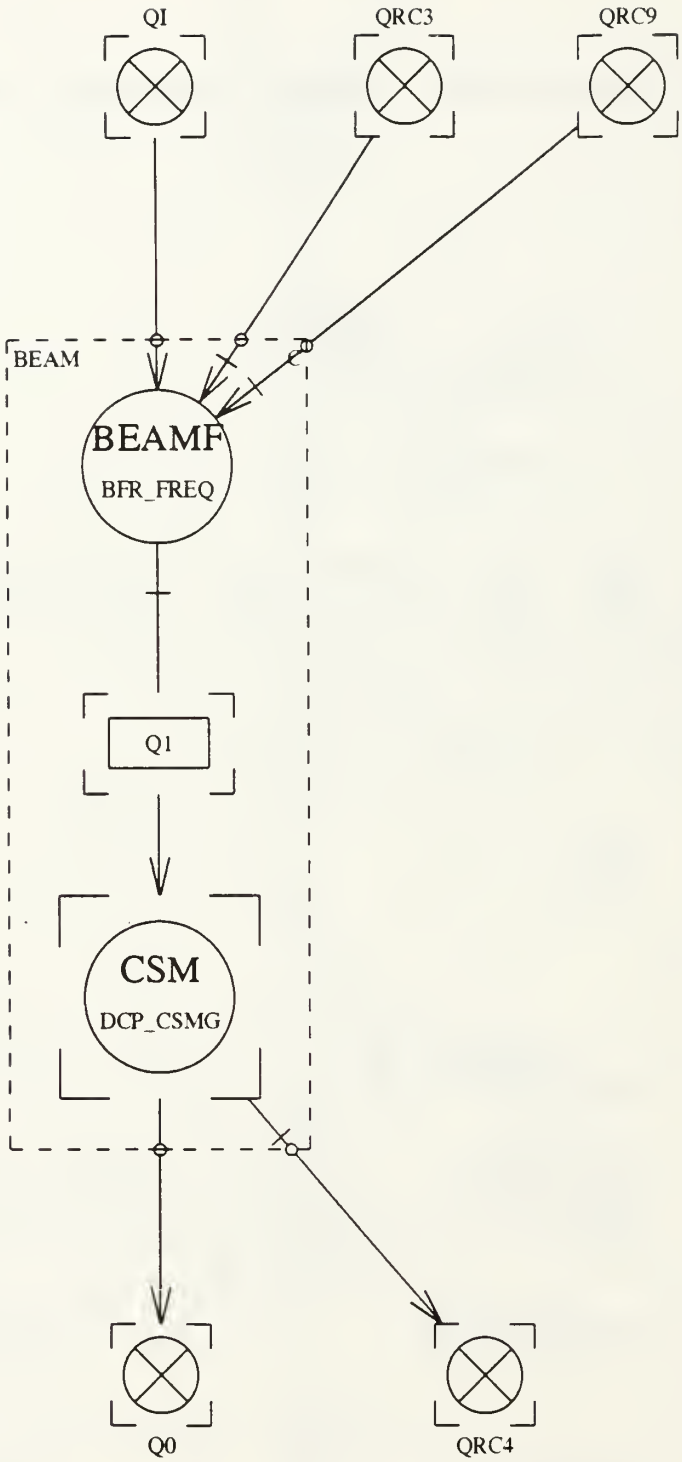
APPENDIX N: RESTRUCTURED ACTIVE GRAPH TOPOLOGY.



Restructured Active Sonobuoy Graph - Root Level Graph



Restructured Active Sonobuoy Graph - Frequency Transform Subgraph



Restructured Active Sonobuoy Graph - Beamformer Subgraph

APPENDIX O: RESTRUCTURED ACTIVE GRAPH SPGN†

```
%%
%% r0a.g: %Z% %P% %I% %G%
%%
%%**************************************************************%%
%% %%
%% GRAPH 'ACT_PROC ' %%
%% SPGN generated from GRED %%
%% Thu May 13 21:24:57 1993 %%
%% %%
%%**************************************************************%%
%GRAPH(ACT_PROC%% GRAPH SPGN FOR 'ACTIVE_PROCESS' OF
SONOBUOY BENCHMARK
 GIP = N: INT,%% INPUT BATCH SIZE
 NF: INT,%% NUMBER OF FREQUENCY CELLS
 NC: INT,%% NUMBER OF INPUT CHANNELS
 NB: INT,%% NUMBER OF BEAMS
 NB1: INT,%% NUMBER OF PEAK PICK BANDS
 NFFT: INT,%% FFT SIZE
 NAVG: INT,%% INTEGRATION TIME
 NW: INT,%% NUMBER OF FREQUENCY WEIGHTS
 NLEVEL: INT,%% NUMBER OF UANTIZATION LEVELS
 TS: INT,%% BANCSHIFT TABLE SIZE
 RATE: DFLOAT,%% INPUT SAMPLING RATE
 FC: DFLOAT,%% BANDSHIFT FREQUENCY
 RFFT: INT,%% FFT REDUNDANCY
 C: DFLOAT,%% CLIPPING CONSTANT
 R: DFLOAT,%% REPLACEMENT CONSTANT
 WIDTH: INT,%% WINDOW WIDTH
 TH: DFLOAT,%% DETECTION THRESHOLD
 IA: INT%% AVLV INDEX
 VAR = AVLV: INT ARRAY(8),%% LSR VALVE
 [1..NB]BWEIGHT: CFLOAT ARRAY(NF,NC)%% BEAMFORMING WEIGHTS
 INPUTQ = [1..NC]Q1: FIXED(0)%% FAMILY OF ACTIVE INPUTS
 OUTPUTQ = CELL: INT V_ARRAY(NF*NB),%% CONCATENATED BIN NUM-
```

† Generated from modified files received from AT&T Bell Laboratories, Whippany, N.J.

```
BERS
 AMP: FLOAT V_ARRAY(NF*NB),%% CONCATENATED AMPLITUDES
 LEVEL: INT,%% CONCATENATED LEVELS
 INDEX: INT%% CONCATENATED INDICES
)


%% DECLARATIONS section (%GIP, %VAR, %QUEUE)

%GIP([1..NB]NB1C: INT
 INITIALIZE TO NB1)

%GIP(TVECTOR: FLOAT ARRAY(NLEVEL-1)%% QUANTIZATION THRESH-
OLDS
 INITIALIZE TO {1.E0,1.2E0,1.4E0,1.8E0,2.2E0,
 4.E0,6.E0})

%GIP(B: INT ARRAY(NF)%% BIN ADDRESS VECTOR
 INITIALIZE TO {512 OF 1})

%GIP([1..4]COEFF: DFLOAT ARRAY(5)%% IIR FILTER COEFFICIENTS
 INITIALIZE [1]COEFF TO {0.125E0,.1952965E0,.125E0,.362915E0,-.171509E0}
 INITIALIZE [2]COEFF TO {.25E0,.1297913E0,.25E0,.178711E0,-.736206E0}
 INITIALIZE [3]COEFF TO {0.125E0,.1952965E0,.125E0,.362915E0,-.171509E0}
 INITIALIZE [4]COEFF TO {.25E0,.1297913E0,.25E0,.178711E0,-.736206E0})

%GIP(FWEIGHT: FLOAT ARRAY(NF,NW)%% FREQUENCY WEIGHTS
 INITIALIZE TO {512*5 OF 1.0E0})

%GIP(N1: INT
 INITIALIZE TO 1024)

%GIP(N2: INT
 INITIALIZE TO 512)

%GIP(W: FLOAT ARRAY(NF,NW)
 INITIALIZE TO {NF*NW OF 1.0E0})

%VAR(R11: INT
 INITIALIZE TO 8)

%VAR(N11: INT
 INITIALIZE TO 1024)
```

```
%QUEUE(CNTR: INT%% FORMAT/PACK FEEDBACK COUNTER
 INITIALIZE TO 0,0,0)

%QUEUE([1..NC]Q2: CFLOAT)%% FAMILY OF NC SPECTRA
%QUEUE([1..16]Q4A: CFLOAT)%% IIR FILTER OUT
%QUEUE(Q4AA: INT)
%QUEUE([1..11]Q4B: INT)
%QUEUE(Q4C: INT)
%QUEUE([1..3]Q4D: INT)
%QUEUE(Q5A: INT)
%QUEUE([1..11]Q5B: INT)
%QUEUE(Q5C: INT)
%QUEUE([1..3]Q5D: INT)
%QUEUE(Q6A: FLOAT V_ARRAY(NF))
%QUEUE([1..11]Q6B: FLOAT V_ARRAY(NF))
%QUEUE(Q6C: FLOAT V_ARRAY(NF))
%QUEUE([1..3]Q6D: FLOAT V_ARRAY(NF))
%QUEUE(Q7A: INT V_ARRAY(NF))
%QUEUE([1..11]Q7B: INT V_ARRAY(NF))
%QUEUE(Q7C: INT V_ARRAY(NF))
%QUEUE([1..3]Q7D: INT V_ARRAY(NF))
%QUEUE([1..11]Q3B: FLOAT)
%QUEUE(Q3C: FLOAT)
%QUEUE([1..3]Q3D: FLOAT)
%QUEUE(Q3A: FLOAT)
%QUEUE([1..16]RC1: TRIGGER
 INITIALIZE [1..4]RC1 TO 9
 INITIALIZE [5..12]RC1 TO 8
 INITIALIZE [13..16]RC1 TO 20)

%QUEUE([1..4]RC2: TRIGGER
 INITIALIZE [1..4]RC2 TO 10)

%QUEUE([1..1]RC3: TRIGGER
 INITIALIZE [1..1]RC3 TO 0)

%QUEUE([1..16]RC4: TRIGGER
 INITIALIZE [1..7]RC4 TO 20
 INITIALIZE [8..15]RC4 TO 0
 INITIALIZE [16..16]RC4 TO 40)

%QUEUE([1..11]RC5: TRIGGER
```

```
 INITIALIZE [1..3]RC5 TO 20
 INITIALIZE [4..7]RC5 TO 0
 INITIALIZE [8..11]RC5 TO 20)

%QUEUE([1..4]RC6: TRIGGER
 INITIALIZE [1..4]RC6 TO 0)

%QUEUE([1..12]RC7: TRIGGER
 INITIALIZE [1..4]RC7 TO 0
 INITIALIZE [5..12]RC7 TO 20)

%QUEUE([1..3]RC8: TRIGGER
 INITIALIZE [1..3]RC8 TO 0)

%QUEUE([1..1]RC9: TRIGGER
 INITIALIZE [1..1]RC9 TO 0)


%% TOPOLOGY section (%NODE, %SUBGRAPH)

%NODE(CONCAT4PRIMITIVE = DFC_VCAT
 PIP_IN = [16..16]RC4THRESHOLD = 48CONSUME = 8
 PRIM_IN = NB,1,1,1,
 FAMILY[Q7A,[1..11]Q7B,Q7C,[1..3]Q7D]THRESHOLD = 1
 PRIM_OUT = UNUSED,CELL)

%NODE(CONCAT3PRIMITIVE = DFC_VCAT
 PIP_IN = [14..14]RC4THRESHOLD = 48CONSUME = 8
 PRIM_IN = NB,1,1,1,
 FAMILY[Q6A,[1..11]Q6B,Q6C,[1..3]Q6D]THRESHOLD = 1
 PRIM_OUT = UNUSED,AMP)

%NODE(GRAMOUTPRIMITIVE = DFC_FMTPK
 PIP_IN = [13..13]RC4THRESHOLD = 48CONSUME = 8
 PRIM_IN = NB1,NB,130,600,6,
 20,30,1,1,
 CNTRTHRESHOLD = 3,
 FAMILY[Q5A,[1..11]Q5B,Q5C,[1..3]Q5D]THRESHOLD = NB1
 PRIM_OUT = CNTR,LEVEL)

%NODE(CONCAT1PRIMITIVE = DFC_CAT
 PIP_IN = [15..15]RC4THRESHOLD = 48CONSUME = 8
 PRIM_IN = NB,1,[1..NB]NB1C,
```

```
FAMILY[Q4AA,[1..11]Q4B,Q4C,[1..3]Q4D]THRESHOLD = NB1
PRIM_OUT = INDEX)

%SUBGRAPH(BEAMFORMGRAPH = FREQ_BEAMFORM
GIP = NF,NC,NB
VAR = [1..NB]BWEIGHT
INPUTQ = [1..NC]Q2,[1..1]RC3,[1..1]RC9
OUTPUTQ = FAMILY[Q3C,Q3A,[1..11]Q3B,[1..3]Q3D],[1..16]RC4)

%SUBGRAPH([I=2..4]FTRANSFORMGRAPH = FREQ_TRANSFORM
GIP = N,TS,RATE,NFFT,NF,
NW,FWEIGHT
VAR = FC,[1..4]COEFF,RFFT,B
INPUTQ = [(I-1)*4+1..(I-1)*4+4]Q1,[I]RC2
OUTPUTQ = [(I-1)*4+1..(I-1)*4+4]Q4A)

%NODE([I=2..4]FQT2PRIMITIVE = SCH_FQT2
PIP_IN = R11,N11,
[I]RC1THRESHOLD = 1
PRIM_IN = N1,N1,N2,0,1,
B,NW,N2,W,
[I]Q4ATHRESHOLD = 1920VARIABLE CONSUME = 8*(N11/R11)
PRIM_OUT = [I]Q2)

%NODE([I=1..11]DETECTPRIMITIVE = SCH_ACT
PIP_IN = AVLV(IA),
[I]RC4THRESHOLD = 72CONSUME = 8,
[I]RC5THRESHOLD = 10,
[I+1..I+1]RC7THRESHOLD = 10CONSUME = 1
PRIM_IN = NF,NAVG,UNUSED,UNUSED,NF/NAVG,
TVECTOR,C,R,WIDTH,1,
TH,UNUSED,
[I]Q3BTHRESHOLD = NF*NAVG
PRIM_OUT = UNUSED,UNUSED,[I]Q4B,
[I]Q5BVARIABLE VALVE = AVLV(IA),
UNUSED,[I]Q6B,[I]Q7B)

%NODE(DET1PRIMITIVE = SCH_ACT
PIP_IN = AVLV(IA),
[1..1]RC7THRESHOLD = 10CONSUME = 1
PRIM_IN = NF,NAVG,UNUSED,UNUSED,NF/NAVG,
TVECTOR,C,R,WIDTH,1,
TH,UNUSED,
```

```
Q3ATHRESHOLD = NF*NAVG
PRIM_OUT = UNUSED,UNUSED,Q4AA,
Q5AVARIABLE VALVE = AVLV(IA),
UNUSED,Q6A,Q7A
PIP_OUT = [1..3]RC8PULSE = 1)

%NODE(DET2PRIMITIVE = SCH_ACT
PIP_IN = AVLV(IA),
[12..12]RC4THRESHOLD = 36CONSUME = 8
PRIM_IN = NF,NAVG,UNUSED,UNUSED,NF/NAVG,
TVECTOR,C,R,WIDTH,1,
TH,UNUSED,
Q3CTHRESHOLD = NF*NAVG
PRIM_OUT = UNUSED,UNUSED,Q4C,
Q5CVARIABLE VALVE = AVLV(IA),
UNUSED,Q6C,Q7C
PIP_OUT = [1..16]RC1PULSE = 1,
[1..4]RC2PULSE = 1)

%NODE([I=1..3]DET3PRIMITIVE = SCH_ACT
PIP_IN = AVLV(IA),
[I]RC8THRESHOLD = 1
PRIM_IN = NF,NAVG,UNUSED,UNUSED,NF/NAVG,
TVECTOR,C,R,WIDTH,1,
TH,UNUSED,
[I]Q3DTHRESHOLD = NF*NAVG
PRIM_OUT = UNUSED,UNUSED,[I]Q4D,
[I]Q5DVARIABLE VALVE = AVLV(IA),
UNUSED,[I]Q6D,[I]Q7D)

%SUBGRAPH([I=1..1]FTRAN1GRAPH = FREQ_TRANSFORM1
GIP = N,TS,RATE,NFFT,NF,
NW,FWEIGHT
VAR = FC,[1..4]COEFF,RFFT,B
INPUTQ = [(I-1)*4+1..(I-1)*4+4]Q1,[I]RC2
OUTPUTQ = [(I-1)*4+1..(I-1)*4+4]Q4A,[I]RC3)

%NODE([I=5..5]FQT2aPRIMITIVE = SCH_FQT2
PIP_IN = R11,N11,
[I]RC1THRESHOLD = 1
PRIM_IN = N1,N1,N2,0,1,
B,NW,N2,W,
[I]Q4ATHRESHOLD = 1920VARIABLE CONSUME = 8*(N11/R11)
```

```
PRIM_OUT = [I]Q2
PIP_OUT = [(I-4)..(I-4)]RC9PULSE = 8)

%NODE([I=6..8]FQT2bPRIMITIVE = SCH_FQT2
PIP_IN = R11,N11,
[I]RC1THRESHOLD = 1
PRIM_IN = N1,N1,N2,0,1,
B,NW,N2,W,
[I]Q4ATHRESHOLD = 1920VARIABLE CONSUME = 8*(N11/R11)
PRIM_OUT = [I]Q2)

%NODE([I=9..9]FQT2cPRIMITIVE = SCH_FQT2
PIP_IN = R11,N11,
[I]RC1THRESHOLD = 1
PRIM_IN = N1,N1,N2,0,1,
B,NW,N2,W,
[I]Q4ATHRESHOLD = 1920VARIABLE CONSUME = 8*(N11/R11)
PRIM_OUT = [I]Q2
PIP_OUT = [(I-9)+1..(I-9)+12]RC7PULSE = 1)

%NODE([I=1..1]FQT2hPRIMITIVE = SCH_FQT2
PIP_IN = R11,N11,
[I]RC1THRESHOLD = 1
PRIM_IN = N1,N1,N2,0,1,
B,NW,N2,W,
[I]Q4ATHRESHOLD = 1920VARIABLE CONSUME = 8*(N11/R11)
PRIM_OUT = [I]Q2
PIP_OUT = [(I-1)+1..(I-1)+4]RC6PULSE = 1)

%NODE([I=10..12]FQT2ePRIMITIVE = SCH_FQT2
PIP_IN = R11,N11,
[I]RC1THRESHOLD = 1
PRIM_IN = N1,N1,N2,0,1,
B,NW,N2,W,
[I]Q4ATHRESHOLD = 1920VARIABLE CONSUME = 8*(N11/R11)
PRIM_OUT = [I]Q2)

%NODE([I=13..13]FQT2fPRIMITIVE = SCH_FQT2
PIP_IN = R11,N11,
[I]RC1THRESHOLD = 1,
[(I-13)+1..(I-13)+1]RC6THRESHOLD = 1
PRIM_IN = N1,N1,N2,0,1,
B,NW,N2,W,
```

```
[I]Q4ATHRESHOLD = 1920VARIABLE CONSUME = 8*(N11/R11)
PRIM_OUT = [I]Q2
PIP_OUT = [(I-12)..(I-2)]RC5PULSE = 10)

%NODE([I=14..16]FQT2gPRIMITIVE = SCH_FQT2
PIP_IN = R11,N11,
[I]RC1THRESHOLD = 1,
[(I-14)+2..(I-14)+2]RC6THRESHOLD = 1
PRIM_IN = N1,N1,N2,0,1,
B,NW,N2,W,
[I]Q4ATHRESHOLD = 1920VARIABLE CONSUME = 8*(N11/R11)
PRIM_OUT = [I]Q2)


%ENDGRAPH


%%*************************************************************%%
%% %%
%% GRAPH 'FREQ_BEAMFORM ' %%
%% SPGN generated from GRED %%
%% Thu May 13 21:24:57 1993 %%
%% %%
%%*************************************************************%%
%%
%%SPGN FOR 'FREQ_BEAMFORM' OF SONOBUOY BENCHMARK
%%CALLED BY ACTIVE_PROCESS
%%
%GRAPH(FREQ_BEAMFORM
GIP = NF: INT,%% NUMBER OF FREQUENCY BINS
NC: INT,%% NUMBER OF CHANNELS
NB: INT%% NUMBER OF BEAMS
VAR = [1..NB]W: CFLOAT ARRAY(NF,NC)%% BEAMFORMING WEIGHTS
INPUTQ = [1..NC]QI: CFLOAT,%% FAMILY OF NC SPECTRA
[1..1]QRC3: TRIGGER,
[1..1]QRC9: TRIGGER
OUTPUTQ = [1..NB]Q0: FLOAT,%% FAMILY OF BEAM MAGNITUDES
[1..16]QRC4: TRIGGER
)


%% DECLARATIONS section (%GIP, %VAR, %QUEUE)
```

```
%% TOPOLOGY section (%NODE, %SUBGRAPH)

%NODE(BEAMFORMERPRIMITIVE = SCH_FREQ
 PIP_IN = [1..1]QRC3THRESHOLD = 17CONSUME = 1,
 [1..1]QRC9THRESHOLD = 1CONSUME = 1
 PRIM_IN = NF,NC,NB,[1..NB]W,.961E0,
 .398E0,
 [1..NC]QITHRESHOLD = NF
 PRIM_OUT = [1..NB]Q0
 PIP_OUT = [1..NB]QRC4PULSE = 1)


%ENDGRAPH


%%*************************************************************%%
%% %%
%% GRAPH 'FREQ_TRANSFORM ' %%
%% SPGN generated from GRED %%
%% Thu May 13 21:24:57 1993 %%
%% %%
%%*************************************************************%%
%%
%%SPGN FOR 'FREQ_TRANSFORM' OF SONOBUOY BENCHMARK
%%CALLED BY ACTIVE_PROCESS
%%
%GRAPH(FREQ_TRANSFORM
 GIP = N: INT,%% BANDSHIFT/FILTER BATCH SIZE
 TS: INT,%% BANDSHIFT TABLE SIZE
 FS: DFLOAT,%% INPUT SAMPLING RATE
 N1: INT,%% FFT SIZE
 N2: INT,%% NUMBER OF OUTPUT BINS
 NW: INT,%% NUMBER OF WEIGHTS PER BIN
 W: FLOAT ARRAY(N2,NW)%% FREQUENCY WEIGHTS
 VAR = FC: DFLOAT,%% BANDSHIFT FREQUENCY
 [1..4]COEFF: DFLOAT ARRAY(5),%% IIR FILTER COEFFICIENTS
 R: INT,%% FFT REDUNDANCY
 B: INT ARRAY(N2)%% BIN ADDRESS VECTOR
 INPUTQ = [1..4]QI: FIXED(0),
 QRC2: TRIGGER
 OUTPUTQ = [1..4]Q40: CFLOAT
)
```

```
%% DECLARATIONS section (%GIP, %VAR, %QUEUE)

%QUEUE([1..4]Q2: INT
 INITIALIZE TO 0)

%QUEUE([1..4]QF_1: DCFLOAT
 INITIALIZE TO 4 OF <0.E0,0.E0>)

%QUEUE([1..4]QF_3: DCFLOAT
 INITIALIZE TO 4 OF <0.E0,0.E0>)

%QUEUE([1..4]QF_4: DCFLOAT
 INITIALIZE TO 4 OF <0.E0,0.E0>)

%QUEUE([1..4]QF_2: DCFLOAT
 INITIALIZE TO 4 OF <0.E0,0.E0>)


%% TOPOLOGY section (%NODE, %SUBGRAPH)

%NODE(FQT1PRIMITIVE = SCH_FQT1
 PIP_IN = QRC2THRESHOLD = 1
 PRIM_IN = N,0,TS,FC,FS,
 [1..4]Q2THRESHOLD = 1,
 1,1,1,2,[1]COEFF,
 [2]COEFF,[3]COEFF,[4]COEFF,0,
 [1..4]QITHRESHOLD = N,
 [1..4]QF_1THRESHOLD = 4,
 [1..4]QF_3THRESHOLD = 4,
 [1..4]QF_4THRESHOLD = 4,
 [1..4]QF_2THRESHOLD = 4
 PRIM_OUT = [1..4]Q40,[1..4]Q2,[1..4]QF_1,[1..4]QF_3,[1..4]QF_4,
 [1..4]QF_2)


%ENDGRAPH


%%*************************************************************%%
%% %%
%% GRAPH 'FREQ_TRANSFORM1 ' %%
```

```
%% SPGN generated from GRED %%
%% Thu May 13 21:24:57 1993 %%
%% %%
%%**********************************************************%%
%%
%%SPGN FOR 'FREQ_TRANSFORM' OF SONOBUOY BENCHMARK
%%CALLED BY ACTIVE_PROCESS
%%
%GRAPH(FREQ_TRANSFORM1
 GIP = N: INT,%% BANDSHIFT/FILTER BATCH SIZE
 TS: INT,%% BANDSHIFT TABLE SIZE
 FS: DFLOAT,%% INPUT SAMPLING RATE
 N1: INT,%% FFT SIZE
 N2: INT,%% NUMBER OF OUTPUT BINS
 NW: INT,%% NUMBER OF WEIGHTS PER BIN
 W: FLOAT ARRAY(N2,NW)%% FREQUENCY WEIGHTS
 VAR = FC: DFLOAT,%% BANDSHIFT FREQUENCY
 [1..4]COEFF: DFLOAT ARRAY(5),%% IIR FILTER COEFFICIENTS
 R: INT,%% FFT REDUNDANCY
 B: INT ARRAY(N2)%% BIN ADDRESS VECTOR
 INPUTQ = [1..4]QI: FIXED(0),
 QRC2: TRIGGER
 OUTPUTQ = [1..4]Q40: CFLOAT,
 QRC3: TRIGGER
)


%% DECLARATIONS section (%GIP, %VAR, %QUEUE)

%QUEUE([1..4]Q2: INT
 INITIALIZE TO 0)

%QUEUE([1..4]QF_1: DCFLOAT
 INITIALIZE TO 4 OF <0.E0,0.E0>)

%QUEUE([1..4]QF_3: DCFLOAT
 INITIALIZE TO 4 OF <0.E0,0.E0>)

%QUEUE([1..4]QF_4: DCFLOAT
 INITIALIZE TO 4 OF <0.E0,0.E0>)

%QUEUE([1..4]QF_2: DCFLOAT
 INITIALIZE TO 4 OF <0.E0,0.E0>)
```

```
%% TOPOLOGY section (%NODE, %SUBGRAPH)

%NODE(FQT1PRIMITIVE = SCH_FQT1
 PIP_IN = QRC2THRESHOLD = 1
 PRIM_IN = N,0,TS,FC,FS,
 [1..4]Q2THRESHOLD = 1,
 1,1,1,2,[1]COEFF,
 [2]COEFF,[3]COEFF,[4]COEFF,0,
 [1..4]QITHRESHOLD = N,
 [1..4]QF_1THRESHOLD = 4,
 [1..4]QF_3THRESHOLD = 4,
 [1..4]QF_4THRESHOLD = 4,
 [1..4]QF_2THRESHOLD = 4
 PRIM_OUT = [1..4]Q40,[1..4]Q2,[1..4]QF_1,[1..4]QF_3,[1..4]QF_4,
 [1..4]QF_2
 PIP_OUT = QRC3PULSE = 8)


%ENDGRAPH
```

LIST OF REFERENCES

1. AT&T Technologies, Report 5885401, *Enhanced Modular Signal Processor (EMSP) Principles of Operation (POPS)*, AT&T Bell Laboratories, 20 March 1990.

2. Rice, M. L., "The Navy's New Standard Digital Signal Processor: The AN/UYS-2," paper presented at the Association of Scientists and Engineers 27th Annual Technical Symposium, 23 May 1990.

3. Gurd, J. R., Kirkhame, C. C., and Watson, I., " The Manchester Prototype Dataflow Computer," *Communications of the ACM*, January 1985.

4. Brobst, S. A., "Organization of an Instruction Scheduling and Token Storage Unit in a Tagged Token Data Flow Machine," *Proceedings of the 1987 International Conference on Parallel Processing*, v. 3. August 1987.

5. Lee, E. A., and Bier, J. C., "Architectures for Statically Scheduled Dataflow," *Journal of Parallel and Distributed Computing*, v. 10, December 1990.

6. Shukla, S. B., Little, B. S., and Zaky, A., "A Compile-time Technique for Controlling Real-time Execution of Task-level Data-flow Graphs," *International Conference on Parallel Processing*, August 1992.

7. AT&T Technologies, *ECOS Workstation Release 5.5 User Manual*, AT&T Bell Laboratories, 1 April 1992.

8. Lee, E. A., and Messerschmitt, D. G., "Static Scheduling of Synchronous Dataflow Programs for Digital Signal Processing," *IEEE Transactions on Computers*, v C-36, No. 1, January 1987.

9. AT&T Technologies, Report IN 48280, *ECOS Workstation Tutorial*, AT&T Bell Laboratories, 30 March 1991.

10. Naval Research Laboratory, *Processing Graph Method Tutorial*, 8 January 1990.

11. AT&T Technologies, Report CDRL Q001, *Enhanced Modular Signal Procesor (EMSP) Application Programmer User Manual*, AT&T Bell Laboratories, 1 August 1990.

12. Naval Research Laboratory, *A PID Generator and Other Performance Improvements for the AN/UYS-2*, 30 August 1989.

13. AT&T Technologies, Report 5885404, AN/UYS-2 Graph Primitives Library - Floating Point, AT&T Bell Laboratories, 17 September 1990.

14. Little, B. S., *A Technique for Predictable Real-time Execution in the AN/UYS-2 Parallel Signal Processing Architecture*, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1991.

15. Akin, C., *A Periodic Input Processing DAta Flow Simulator: PIPDAFS*, Master's Thesis, Naval Postgraduate School, Monterey, California, March 1992.

16. AT&T Technologies, Report CDRL Q001, *Machine Configuration (med) and Simulation Performance Interface (spi) User's Manual*, AT&T Bell Laboratories, 1 August 1992.

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, VA 22314-6145	2
2.	Dudley Knox Library, Code 52 Naval Postgraduate School Monterey, CA 93943-5002	2
3.	Chairman, Code EC Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5000	1
4.	Prof. Shridhar Shukla, Code EC/Sh Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5000	2
5.	Prof. Amr Zaky, Code CS/Za Department of Computer Science Naval Postgraduate School Monterey, CA 93943-5000	1
6.	Mr. David Kaplan Naval Research Laboratory 4555 Overlook Avenue, SW Washington, D.C. 20375-5000	1
7.	Mr. Richard Stevens Naval Research Laboratory 4555 Overlook Avenue, SW Washington, D.C. 20375-5000	1
8.	Commander, Naval Sea Systems Command PMS 428 Naval Sea Systems Command Headquarters Washington, D.C., 20362-5101	1

9. American Telephone and Telegraph Bell Laboratories 1
Attn: Mr. Jerome Uhrig, WH 46243
67 Whippany Road
P.O. Box 903
Whippany, N.J. 07981-0903

10. David P. Swank
2030 Kerr Road
Harleysville, PA 19438 1